Università degli Studi di Salerno

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica applicata

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea in:

Sviluppo Full-Stack di un Gestionale Aziendale Proprietario

Relatore:

Antonio Della Cioppa

Laureando:

Aldo Buongiorno

ANNO ACCADEMICO 2024/2025

Ringraziamenti

```
[INFO] Caricamento ringraziamenti...
[WARN] Nessun contenuto trovato.
[DEBUG] Motivazione:  TROPPA PASQUETTA QUEST'ANNO.
[INFO] Rinviato tutto alla prossima release.
[ERROR] Ringraziamenti non generati.
[EXIT] Codice di uscita:  1
```

Abstract

Nel contesto aziendale moderno, la gestione efficiente delle risorse e dei processi operativi rappresenta un fattore chiave per la competitività e la sostenibilità economica. Questo elaborato descrive l'analisi, la progettazione e lo sviluppo di una sezione di un gestionale aziendale proprietario dedicata al Direct Fulfillment (DF), realizzata per ottimizzare le attività operative di magazzino all'interno di un'azienda attiva nel settore dell'import-export. L'obiettivo del progetto è stato quello di fornire un'interfaccia intuitiva, da utilizzare su tablet, destinata ai dipendenti incaricati delle attività logistiche, con lo scopo di ottimizzare la gestione automatizzata degli ordini, ridurre i costi operativi e migliorare l'efficienza dei processi aziendali, quali il monitoraggio delle transazioni[1], l'efficienza dei flussi di lavoro, la sicurezza nella gestione dei dati e la generazione di report dettagliati, riducendo il margine di errore. Attraverso l'integrazione di queste funzionalità avanzate il sistema ha, quindi, lo scopo di migliorare l'efficienza operativa e fornire uno strumento centralizzato per l'amministrazione aziendale.

La realizzazione del progetto segue le best practices di sviluppo software [8] e prevede l'adozione di metodologie agili per garantire flessibilità e adattabilità alle esigenze del cliente. La piattaforma realizzata segue un approccio full-stack, con Spring Boot [15] per il backend, Angular [14] per il frontend e l'integrazione di REST API (REpresentational State Transfer Application Programming Interface) per la comunicazione tra le componenti. La sicurezza dell'applicazione è garantita tramite JWT (JSON Web Token), mentre per la persistenza dei dati è stato utilizzato un database relazionale con linguaggio PostgreSQL [1]. L'architettura adottata, basata su un modello client-server, ha permesso di garantire modularità, scalabilità e sicurezza. L'interfaccia sviluppata è progettata per semplificare il lavoro dell'operatore,

[1] Nel contesto software, una transazione è un'operazione (o un insieme di operazioni) che modifica lo stato del sistema, è tracciabile, atomica (tutto o niente), consistente, isolata e persistente (ACID).

migliorando l'usabilità e l'efficienza del processo.

I risultati ottenuti confermano come soluzioni software personalizzate possano rappresentare un valore aggiunto e un concreto vantaggio competitivo per le imprese, contribuendo alla digitalizzazione dei processi interni e migliorando l'organizzazione operativa e la tracciabilità delle attività, affermando l'importanza dei gestionali proprietari nello scenario competitivo attuale.

Indice

Elenco delle figure

Introduzione

La digitalizzazione dei processi aziendali è oggi una leva strategica fondamentale per aumentare l'efficienza e la competitività, soprattutto in contesti dinamici come quello dell'import-export. In particolare, le operazioni logistiche e di magazzino rappresentano un ambito in cui l'introduzione di strumenti informatici mirati può apportare notevoli benefici in termini di automazione, tracciabilità, efficienza nella gestione delle risorse e riduzione degli errori.

Durante il mio tirocinio formativo presso un'azienda operante nel settore, Magusa S.r.l, ho potuto osservare da vicino le difficoltà e i costi gestionali legate ad alcuni flussi operativi non completamente digitalizzati e di comprendere da vicino le dinamiche e le esigenze organizzative interne, operative e gestionali di un'azienda attiva nel settore. In particoalre, ho avuto modo di comprendere le limitazioni degli strumenti esistenti e le necessità specifiche che un gestionale aziendale dovrebbe soddisfare per migliorare la produttività e la gestione delle informazioni.

La presente tesi nasce quindi con l'obiettivo di progettare e sviluppare una sezione del gestionale proprietario, focalizzata sulla gestione delle attività di DF. L'intervento ha riguardato in particolare la progettazione e realizzazione di un'interfaccia user-friendly, pensata per essere utilizzata su tablet da parte di operatori addetti al magazzino. Questa interfaccia permette di eseguire in maniera automatizzata alcune operazioni ricorrenti, quali il monitoraggio delle transazioni, la gestione degli ordini e l'autenticazione sicura degli utenti, riducendo sensibilmente il rischio di errori manuali.

Il progetto ha richiesto lo sviluppo della sezione, sia lato frontend che backend, seguendo un approccio full-stack e integrando tecnologie moderne come Spring Boot, Angular, JWT e PostgreSQL, oltre a servizi cloud offerti da Amazon Web Service (AWS) [12]. L'architettura

adottata, basata sul modello client-server e sul paradigma RESTful [9], ha consentito di realizzare un sistema scalabile, sicuro e integrato con l'infrastruttura aziendale già esistente.

Nei capitoli seguenti verranno illustrati: il contesto aziendale e le esigenze analizzate durante il tirocinio; le tecnologie e le scelte progettuali adottate; l'architettura della soluzione sviluppata; e infine una valutazione dei risultati ottenuti e delle possibili evoluzioni future.

CHAPTER

1 Obiettivi e tematiche di ricerca e sviluppo

1.1 Panoramica delle tecnologie utilizzate

La sezione del gestionale sviluppato in questo progetto è stata realizzata utilizzando un insieme di tecnologie moderne e consolidate, con l'obiettivo di garantire scalabilità, modularità e manutenibilità del sistema.

Per la componente backend è stato utilizzato Spring Boot, un framework open-source basato su Java [7], che semplifica la creazione di applicazioni stand-alone robuste, flessibili e facilmente distribuibili. Questo espone un insieme di REST API che permettono al frontend di interagire con le funzionalità del sistema, gestendo la logica di business e accedendo ai dati in modo strutturato e sicuro.

La sicurezza dell'applicazione è stata gestita mediante JWT [10], una soluzione leggera ed efficace per l'autenticazione e l'autorizzazione degli utenti, che consente di implementare meccanismi di accesso basati su token e di proteggere le risorse esposte dal backend.

Per quanto riguarda la persistenza dei dati, è stato utilizzato PostgreSQL, RDBMS (Sistema di Gestione di Basi di Dati Relazionali – *Relational DataBase Management System* –) open-source, affidabile e potente, che supporta lo standard SQL e offre avanzate funzionalità di gestione dei dati. A supporto dell'infrastruttura è stata inoltre integrata la piattaforma AWS, impiegata per l'accesso remoto ai file, in particolare la gestione e l'automatizzazione dei dati degli ordini, oltre che per garantire la scalabilità e l'affidabilità del sistema.

Il frontend è stato realizzato con Angular, framework TypeScript [13] per lo sviluppo di interfacce web dinamiche. Per la progettazione dell'interfaccia utente, lo stile e la gestione della responsività è stato utilizzato Bootstrap [16], rendendo l'applicazione facilmente fruibile anche da dispositivi mobili, in particolare su tablet, contesto in cui si prevede l'utilizzo più frequente della sezione del gestionale sviluppata.

L'intera soluzione segue un'architettura client-server e, adottando un approccio full-stack, consente una piena integrazione tra frontend e backend. Le due componenti comunicano tramite chiamate HTTP (HyperText Transfer Protocol) [4], secondo il paradigma RESTful, garantendo una gestione coerente e strutturata dell'intero ciclo applicativo.

1.2 Obiettivi

Nel 2021 l'azienda ha avviato lo sviluppo di un proprio gestionale, spinta dalla necessità di un'integrazione profonda con gli strumenti forniti direttamente da Amazon, partner con cui collabora attivamente. Tuttavia, l'assenza di soluzioni software in grado di soddisfare pienamente le specifiche esigenze operative ha reso necessario la creazione di un sistema dedicato. Per questo motivo, l'azienda ha optato per la realizzazione di una piattaforma software capace di adattarsi al proprio flusso di lavoro e di rispondere in modo preciso alle richieste del settore di riferimento.

L'intento centrale di questo lavoro è stato lo sviluppo di una specifica sezione del gestionale aziendale proprietario, progettata per essere utilizzata da molteplici dipendenti addetti alla logistica di magazzino attraverso dei dispositivi palmari (tablet), in un contesto operativo legato al DF. Tale sezione è stata personalizzata in base alle esigenze dell'azienda con particolare attenzione all'integrazione con il sistema master già esistente. In particolare si vuole:

- Fornire un'interfaccia accessibile, responsive e user-friendly per l'utilizzo da parte dei magazzinieri del software sui tablet;
- Consentire l'accesso autenticato e sicuro dell'utente;
- Permettere ai magazzinieri di visualizzare e gestire gli ordini in autonomia, monitorandone lo stato e intervenendo secondo necessità operative;

- Segnalare la disponibilità e la presenza degli item nel magazzino, contribuendo all'allineamento dei database aziendali;
- Migliorare la tracciabilità delle operazioni di magazzino, mediante la registrazione dettagliata delle azioni degli utenti e degli stati degli ordini, con l'obiettivo di minimizzare gli errori manuali e aumentare l'affidabilità dell'intero processo operativo.
- Garantire l'accesso remoto ai dati e la scalabilità del sistema grazie all'integrazione con i servizi cloud;
- Favorire l'integrazione futura con altri moduli del gestionale, assicurando flessibilità ed estendibilità del sistema;

L'obiettivo è stato perseguito integrando competenze sia di frontend che di backend in un approccio full-stack, che garantisce flessibilità nello sviluppo e una visione completa dell'intero ecosistema applicativo.

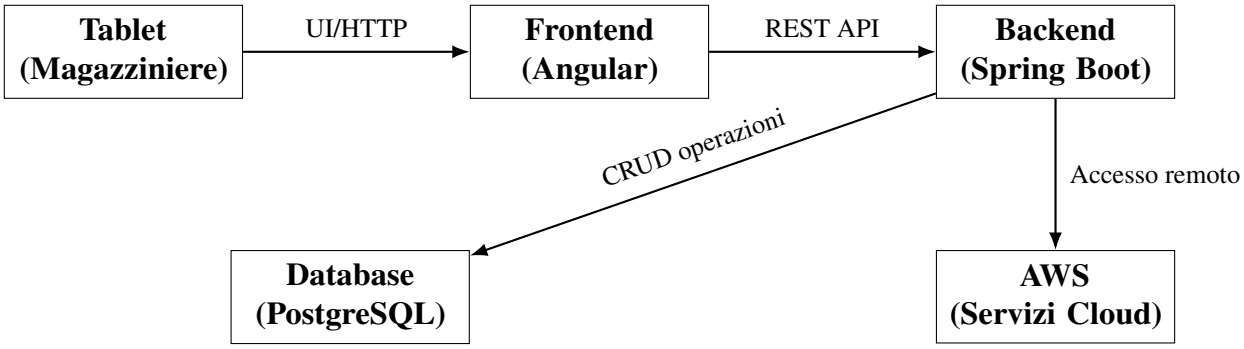


Figura 1.1: Flusso di comunicazione tra i componenti del sistema

1.3 Tematiche di ricerca e sviluppo

Di seguito le diverse tematiche tecniche che hanno guidato le scelte progettuali e architetturali:

- **Modello RESTful:** definizione di un'interfaccia coerente tra frontend e backend, con un'attenzione particolare all'organizzazione semantica delle risorse.
- **Sicurezza e autenticazione:** implementazione di un sistema di autenticazione basato su JWT, per garantire l'accesso sicuro alle risorse e la gestione dei permessi utente.

- **Integrazione cloud:** utilizzo dei servizi AWS per consentire una futura comunicazione tra il tool master e la sezione di riferimento e garantire affidabilità e scalabilità dei dati.
- **Progettazione full-stack:** sviluppo integrato del frontend e del backend, con attenzione alla sincronizzazione dei dati e all'esperienza utente.
- **Gestione del modello dati:** progettazione di uno schema relazionale coerente con i requisiti aziendali e ottimizzazione delle operazioni CRUD (Create,Read,Update,Delete).
- **User Experience (UX):** progettazione di un'interfaccia utente chiara, responsive e intuitiva, in grado di semplificare l'interazione con il sistema anche per utenti non tecnici.

Queste tematiche hanno rappresentato non solo sfide tecniche, ma anche occasioni di apprendimento e crescita professionale, contribuendo alla realizzazione di un prodotto solido, scalabile e orientato alle reali esigenze del contesto aziendale.

CHAPTER

2 Contesto, problema, modello e tecnologie abilitanti

Una volta inseritomi nel contesto aziendale di Magusa, si è proceduto con una disamina del problema identificato, dei modelli architetturali valutati e delle tecnologie selezionate per la progettazione della sezione del gestionale proprietario sviluppata. Di seguito, l'approfondimento di questi temi.

2.1 Il contesto

Magusa S.r.l è un'azienda italiana con sede a Nocera Inferiore, attiva nel settore del commercio all'ingrosso di articoli sportivi, con un forte orientamento all'import-export e una consolidata collaborazione con Amazon.

Nel 2021, Magusa ha avviato lo sviluppo di un sistema gestionale proprietario, attualmente già in produzione in una versione light e correttamente integrato su tutti i marketplace Amazon gestiti dall'azienda. Il sistema, nato per supportare le attività core legate alla piattaforma Amazon — come la gestione degli ordini, delle spedizioni e del magazzino — rappresenta oggi un asset strategico per il consolidamento e l'evoluzione dell'infrastruttura digitale aziendale.

Alla luce dell'espansione delle attività e della necessità di rendere il sistema sempre più adattabile a nuove linee di business, Magusa ha avviato una fase di miglioramento continuo, con l'obiettivo di aumentare il livello di automazione e ottimizzare i costi legati all'infrastruttura cloud.

In particolare, i principali ambiti su cui si concentrano gli sviluppi futuri riguardano:

- L'automazione avanzata nella gestione degli ordini e delle spedizioni;
- L'implementazione e il potenziamento di sistemi di monitoraggio delle performance operative.

In questo contesto si inserisce lo sviluppo della sezione del gestionale dedicata al DF, progettata per migliorare l'operatività degli addetti al magazzino tramite l'utilizzo di dispositivi palmari. Tale componente si pone come obiettivo quello di facilitare le attività logistiche, integrandosi con il sistema esistente e contribuendo all'automazione e alla tracciabilità dei processi.

2.2 Il problema

Il problema identificato riguardava la necessità, da parte di Magusa, di dotarsi di uno strumento software dedicato alla gestione e al monitoraggio delle operazioni di carico e scarico all'interno del magazzino aziendale. Tali attività, fondamentali per il corretto funzionamento della catena logistica, richiedevano una soluzione che fosse al tempo stesso affidabile, intuitiva e facilmente accessibile per il personale incaricato.

Nello specifico, l'applicazione doveva essere installata su dispositivi tablet forniti direttamente dall'azienda ai dipendenti addetti alla movimentazione della merce. Il software avrebbe dovuto guidare gli operatori nella gestione quotidiana degli ordini, permettendo loro di consultare lo stato degli articoli, monitorarne l'avanzamento e aggiornare le informazioni di inventario.

Un aspetto particolarmente rilevante era rappresentato dalla necessità di garantire un elevato livello di sicurezza, sia nell'accesso all'applicazione (tramite meccanismi di autenticazione robusti come JWT), sia nell'esecuzione delle operazioni, in modo da tutelare la correttezza dei dati gestiti e la privacy degli utenti finali. Non da meno vi era anche il tema rigurdante il download e l'upload di file su server esterni come i bucket S3 gestiti da AWS [12]

2.3 Background Tecnologico

Dopo aver fornito, nel primo capitolo [1.1], una panoramica generale delle tecnologie utilizzate, questa sezione si propone di approfondirne alcune, introducendo concetti e strumenti fondamentali al fine di agevolare la comprensione dei capitoli successivi.

2.3.1 MVC

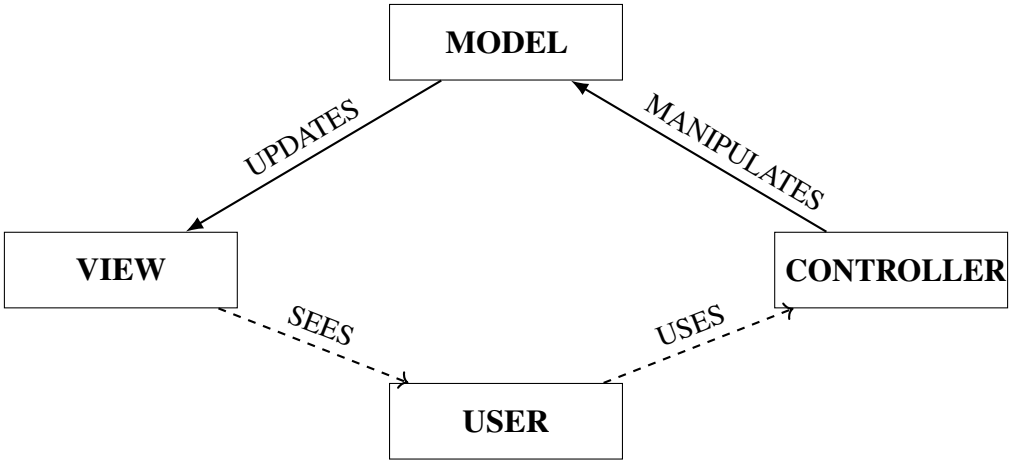


Figura 2.1: Architettura MVC: interazioni tra Model, View, Controller e User

Il pattern architetturale Model-View-Controller (MVC) [11] è ampiamente utilizzato nello sviluppo di applicazioni software per separare le responsabilità logiche e semplificare la gestione del codice. La sua struttura si basa su tre componenti principali, ciascuno con un ruolo specifico:

- **Model**: rappresenta la struttura dei dati e incapsula la logica di accesso e gestione delle informazioni. È responsabile della comunicazione con la base di dati e delle operazioni di lettura e scrittura.
- **View**: si occupa della presentazione delle informazioni all'utente finale. Riceve i dati dal model e li visualizza in modo coerente e comprensibile.
- **Controller**: funge da intermediario tra la view e il model. Gestisce gli input dell'utente, li interpreta e invoca le operazioni appropriate sul model.

Il funzionamento dell'architettura MVC [Figura 2.1] può essere sintetizzato nel seguente ciclo di interazioni:

- L'utente interagisce con l'interfaccia grafica (view), ad esempio inviando una richiesta o compilando un modulo;
- La view trasmette l'input ricevuto al controller;
- Il controller elabora l'input, invoca la logica di business necessaria attraverso il model, ed eventualmente aggiorna lo stato dei dati;
- Il model, a seguito delle modifiche, notifica la view, la quale aggiorna la presentazione in base alle nuove informazioni.

Questa suddivisione delle responsabilità consente una maggiore modularità e manutenibilità del codice, facilitando lo sviluppo e l'estensione dell'applicazione.

In una tipica applicazione web, i tre componenti possono essere associati ai seguenti livelli:

- Il Model è rappresentato dalla struttura dati e dalle entità persistite nel database;
- Il Controller è generalmente implementato nel backend (nel caso specifico tramite Spring Boot), ed è accessibile tramite REST API;
- La View è gestita nel frontend (in questo caso Angular), e si occupa dell'interazione con l'utente.

L'adozione del pattern MVC nella soluzione sviluppata ha consentito di mantenere una chiara separazione tra la logica di presentazione, la gestione dell'input e la manipolazione dei dati, garantendo maggiore chiarezza progettuale e una più semplice scalabilità futura del sistema.

2.3.2 REST API

Le REST API (REpresentational State Transfer Application Programming Interface) rappresentano uno dei paradigmi più diffusi per la realizzazione di interfacce di comunicazione tra sistemi distribuiti, in particolare tra client e server.

REST è uno stile architetturale basato su un insieme di vincoli progettuali che si fondano sull'uso del protocollo HTTP (HyperText Transfer Protocol) e su un'interazione stateless [2.3.2]

tra le componenti. Le API RESTful consentono al client di inviare richieste al server per accedere, modificare o cancellare risorse, rappresentate tipicamente in formato JSON (JavaScript Object Notation) [5] o XML (Extensible Markup Language) [6].

I concetti fondamentali su cui si basa REST sono:

- **Risorse**: ogni entità (es. ordine, prodotto, utente) è rappresentata da un identificatore univoco, normalmente espresso tramite un URI (Uniform Resource Identifier) [3].
- **Verbi HTTP**: le operazioni sulle risorse sono mappate su metodi standard del protocollo HTTP:
  - *GET*: per ottenere una risorsa;
  - *POST*: per creare una nuova risorsa;
  - *PUT*: per aggiornare una risorsa esistente;
  - *DELETE*: per rimuovere una risorsa.
- **Statelessness**: ogni richiesta contiene tutte le informazioni necessarie per essere processata; il server non mantiene alcuno stato tra una richiesta e l'altra.
- **Rappresentazione delle risorse**: le risorse vengono scambiate tra client e server in un formato standardizzato (solitamente JSON), permettendo interoperabilità e semplicità di integrazione.

Nel contesto del progetto realizzato, le REST API costituiscono il punto di comunicazione tra il frontend Angular e il backend Spring Boot. Attraverso esse, la vista può inviare richieste al controller, ricevere risposte formattate e aggiornare l'interfaccia utente in base ai dati ottenuti.

L'utilizzo di REST ha consentito di definire un'interfaccia ben strutturata e facilmente estendibile, promuovendo la separazione tra client e server e rendendo possibile la futura integrazione con altri moduli del gestionale o applicazioni esterne.

Figura 2.2: Esempio di funzionamento di una REST API

2.3.3 Database relazionali

Nel contesto dello sviluppo di un sistema gestionale, il dato rappresenta il fulcro dell'intero ecosistema applicativo. Ogni funzionalità, processo o interfaccia ha come obiettivo ultimo quello di generare, manipolare, visualizzare o archiviare dati in modo coerente, sicuro e persistente. Proprio per questo motivo, la progettazione di una struttura dati solida ed efficiente costituisce una fase fondamentale dell'intero ciclo di sviluppo.

I database relazionali offrono un modello consolidato per la rappresentazione strutturata delle informazioni, basato su tabelle collegate tra loro tramite relazioni logiche. Tale approccio consente non solo di organizzare i dati in modo ordinato, ma anche di garantire integrità, consistenza e tracciabilità, aspetti imprescindibili in un contesto aziendale in cui la qualità delle informazioni gestite ha un impatto diretto sull'operatività. L'integrità dei dati è garantita attraverso vincoli come le chiavi primarie, le chiavi esterne e le restrizioni di unicità. Tale impostazione favorisce, inoltre, la coerenza tra le entità e fornisce una rappresentazione fedele delle relazioni logiche tra i dati.

Per rendere possibile l'interazione con una base di dati, si fa ricorso al linguaggio SQL (Structured Query Language), utilizzato per definire la struttura delle tabelle, interrogarne i contenuti e gestire la persistenza dei dati.

Linguaggio SQL

Le principali operazioni del linguaggio SQL si distinguono in:

- **DDL (Data Definition Language)**: per la definizione delle tabelle, degli schemi e dei vincoli (CREATE, ALTER, DROP);
- **DML (Data Manipulation Language)**: per l'inserimento, modifica o eliminazione dei dati (INSERT, UPDATE, DELETE);
- **DQL (Data Query Language)**: per l'interrogazione dei dati (SELECT);
- **DCL (Data Control Language)**: per la gestione dei permessi e della sicurezza (GRANT, REVOKE).

Nel progetto descritto in questo elaborato, è stato utilizzato l'RDBMS (Sistema di Gestione di Basi di Dati Relazionali – *Relational DataBase Management System* –) PostgreSQL, apprezzato per la sua affidabilità, conformità agli standard e supporto per funzionalità avanzate, tra cui l'integrità referenziale, le transazioni ACID (Atomicity, Consistency, Isolation, Durability) e l'estendibilità mediante funzioni personalizzate. L'impiego di PostgreSQL ha permesso di modellare efficacemente le entità del dominio applicativo (come ordini, utenti e articoli), garantendo coerenza e integrità dei dati e supportando le operazioni fondamentali richieste dal backend per la gestione della logica di business.

CHAPTER

3 Progettazione e implementazione del prototipo

In questo capitolo verranno analizzate nel dettaglio la progettazione e l'implementazione del prototipo sviluppato. Si illustrerà la finalità operativa della sezione realizzata, fornendo una visione complessiva dell'architettura del sistema, una descrizione della struttura del database [3.3], delle tecnologie abilitanti impiegate [3.4] e, infine, un approfondimento sull'implementazione [3.6] concreta delle funzionalità principali.

3.1 Approccio metodologico allo sviluppo

Per la progettazione e lo sviluppo dell'applicazione è stata adottata una delle metodologie appartenenti alla famiglia Agile, nota come XP (Extreme Programming) [2]. Si tratta di un modello iterativo e incrementale, che pone particolare attenzione alla qualità del codice, alla rapida risposta ai cambiamenti nei requisiti e alla collaborazione costante con il cliente. La metodologia XP prevede la definizione di ruoli specifici:

- il Customer, ovvero il cliente (in questo caso Magusa), che ha il compito di stabilire le specifiche, assegnare le priorità e definire i criteri di verifica del sistema;
- il Developer, lo sviluppatore (il sottoscritto), responsabile dell'implementazione delle funzionalità;
- il Tester, incaricato di verificare il corretto funzionamento del sistema attraverso prove di test;

- il Tracker, che monitora l'andamento dello sviluppo in termini di tempi, avanzamento e coerenza rispetto agli obiettivi prefissati.

Nel contesto di questo progetto, i ruoli di Tester e Tracker sono stati svolti dal programmatore interno di Magusa, che ha supervisionato e supportato il lavoro durante l'intero periodo di tirocinio. I principi cardine dell'Extreme Programming che hanno guidato l'intero processo possono essere così sintetizzati:

- Comunicazione: favorire un dialogo continuo tra sviluppatore e cliente, rendendo quest'ultimo partecipe del processo produttivo e decisionale.
- Feedback: sia da parte del cliente, grazie alla comunicazione diretta e costante, sia dal sistema stesso, attraverso l'esecuzione regolare di test di integrazione volti a verificare la correttezza e l'efficacia delle funzionalità.
- Semplicità: mantenere il design del codice quanto più possibile semplice e standardizzato, così da agevolare la manutenzione, aumentare la robustezza e ridurre la necessità di produrre documentazione eccessivamente dettagliata.

L'applicazione di questa metodologia ha permesso uno sviluppo snello, flessibile e costantemente allineato alle reali esigenze aziendali, facilitando il confronto continuo con il personale di Magusa e garantendo al contempo una maggiore reattività ai cambiamenti.

3.2 I requisiti

L'elicitazione dei requisiti rappresenta una fase fondamentale nel processo di sviluppo software, in quanto consente di individuare in modo chiaro e strutturato ciò che il sistema deve realizzare e le caratteristiche che deve possedere. In questa sezione vengono analizzati i requisiti emersi durante il periodo di tirocinio presso Magusa, in stretta collaborazione con il personale tecnico aziendale. Tali requisiti sono stati suddivisi in due categorie principali:

- **Requisiti funzionali**: definiscono le funzionalità che il sistema deve offrire, ovvero i comportamenti attesi in risposta alle azioni dell'utente o ad altri eventi. Descrivono cosa il sistema è in grado di fare, come ad esempio la gestione degli ordini, l'autenticazione degli utenti o la consultazione dello stato degli articoli.

- **Requisiti non funzionali**: riguardano le qualità generali del sistema e i vincoli da rispettare, come la sicurezza, la scalabilità, la manutenibilità o le prestazioni. Non descrivono una funzionalità specifica, ma caratteristiche trasversali che influenzano l'esperienza d'uso e l'affidabilità complessiva.

L'identificazione accurata di entrambe le classi di requisiti ha guidato le scelte progettuali e implementative, contribuendo alla realizzazione di una soluzione coerente con gli obiettivi aziendali e le esigenze operative specifiche del contesto.

3.2.1 Requisiti funzionali

- **RF01**: Il sistema deve essere dotato di un database relazionale per la persistenza strutturata dei dati.
- **RF02**: L'utente deve poter accedere al sistema inserendo nome utente e password tramite l'interfaccia installata sul tablet.
- **RF03**: Il sistema deve verificare le infromazioni confrontandole con il valore associato all'utente nel database.
- **RF04**: In caso di password o nome utente errato, il sistema deve negare l'accesso e fornire un messaggio di errore all'utente.
- **RF05**: In caso di autenticazione corretta, il sistema deve generare e restituire un token JWT, da utilizzare per le successive richieste autenticate.
- **RF06**: Tutte le operazioni relative agli ordini devono essere consentite solo se l'ordine risulta presente nella tabella `orders` del database.
- **RF07**: Il sistema deve permettere all'utente di creare ordini solo se:
  - il magazzino in cui si trovano gli articoli è presente nella tabella `warehouse`;
  - l'utente che sta creando gli ordini ha il campo di `admin` della tabella `users` settato come `true`;
- **RF08**: Il sistema deve permettere all'utente di accettare o rifiutare gli ordini;
- **RF09**: Il sistema deve permettere all'utente di rifiutare gli ordini solo se viene inserita una motivazione;

- **RF10**: Il sistema deve permettere all'utente di consultare informazioni sugli articoli presenti in magazzino, in particolare la quantità disponibile per ciascuno.
- **RF11**: Il sistema deve consentire all'utente di effettuare il logout, invalidando il token JWT associato alla sessione corrente.
- **RF12**: Il sistema deve registrare ogni operazione effettuata dall'utente al fine di garantirne la tracciabilità.
- **RF13**: Il sistema deve permettere l'aggiornamento dello stato degli ordini (creato, accettato, rifiutato, spedito).

3.2.2 Requisiti non funzionali

- **RNF01 – Prestazioni**: Il sistema deve garantire tempi di risposta adeguati alle operazioni richieste dall'utente, in particolare durante l'autenticazione e la gestione degli ordini mantenendo una latenza minima.
- **RNF02 – Scalabilità**: Il sistema deve essere in grado di gestire un numero crescente di utenti e operazioni contemporanee senza degrado significativo delle prestazioni, anche in presenza di un elevato volume di richieste.
- **RNF03 – Affidabilità**: L'applicazione deve garantire un funzionamento stabile e continuo, con un elevato grado di disponibilità, specialmente durante le ore operative di magazzino.
- **RNF04 – Sicurezza**: Il sistema deve garantire la protezione dei dati e delle credenziali utente tramite meccanismi di autenticazione sicura (JWT) e accesso controllato alle risorse.
- **RNF05 – Manutenibilità**: L'architettura del sistema deve garantire un'elevata manutenibilità, grazie a una struttura del codice chiara e modulare, e a una progettazione orientata a semplificare interventi futuri di aggiornamento o estensione.
- **RNF06 – Usabilità**: L'interfaccia utente deve essere semplice, chiara e ottimizzata per l'utilizzo tramite tablet, in modo da risultare facilmente accessibile anche a operatori non esperti in ambito informatico.

- **RNF07 – Portabilità**: L'applicazione frontend deve essere compatibile con dispositivi mobili (in particolare tablet Android e iPad), garantendo la corretta visualizzazione e interazione su schermi touch.

3.3 Progettazione della base di dati

La base di dati su cui si fonda il prototipo è stata progettata e implementata nel backend tramite JPA (Java Persistence API), che consente di mappare le classi Java come entità persistenti, traducendole automaticamente in tabelle relazionali nel database. Ogni entità è identificata da una chiave primaria generata automaticamente (`id`), che consente di tracciare univocamente ogni record inserito. La struttura del database è stata modellata per rispondere in modo coerente ai requisiti funzionali dell'applicazione e per garantire la compatibilità futura con il gestionale già in uso presso Magusa, da cui sono state riprese alcune tabelle e convenzioni strutturali.

Il diagramma ER può essere logicamente suddiviso in tre macroaree principali:

- Gestione utenti e autorizzazioni;
- Gestione degli ordini e degli articoli;
- Storico degli stati degli ordini;

3.3.1 Gestione utenti e autorizzazioni

Le entità `users` e `users_warehouses` rappresentano il cuore del sistema di autenticazione e gestione dei permessi. La tabella `users` [Figura 4.1] contiene i dati essenziali per l'identificazione e l'accesso (`username`, `password`, `scadenza della licenza`, ecc.). La tabella `users_warehouses` stabilisce il legame tra un utente e uno o più magazzini a cui è associato, registrando anche la data di attivazione o eventuale distacco (`attached_at`, `detached_at`). Questo schema consente di abilitare meccanismi flessibili di autorizzazione legati alla posizione fisica o operativa del dipendente.

3.3.2 Gestione degli ordini e degli articoli

Il nucleo informativo relativo al magazzino è rappresentato dalle entità `orders` [Figura 4.2] e `order_items` [Figura 4.3]. La tabella `orders` registra gli ordini gestiti dal sistema, includendo informazioni come codice identificativo, indirizzo e corriere di consegna, date e stato dell'ordine, oltre all'associazione al magazzino di riferimento. La tabella `order_items` specifica i singoli articoli collegati a ciascun ordine, con informazioni dettagliate come codice EAN[1], descrizione, SKU[2], marca, colore, taglia, quantità, codice articolo e quantità per confezione. Questa struttura consente di gestire in modo preciso le operazioni di carico e scarico in magazzino, così come la tracciabilità del contenuto degli ordini.

3.3.3 Storico degli stati degli ordini

La tabella `orders_status_history` è stata progettata per garantire la tracciabilità completa degli aggiornamenti sugli ordini. Essa registra ogni modifica di stato (ad esempio: "accettato", "rifiutato", "spedito"), insieme a una nota in caso di rifiuto, alla data di creazione del record e all'utente che ha effettuato l'operazione. In questo modo è possibile ricostruire in qualsiasi momento il flusso di gestione di un ordine e risalire ai responsabili delle operazioni.

Nel complesso, la struttura del database è stata pensata per offrire:

- coerenza tra gli oggetti gestiti dall'applicazione (utenti, articoli, ordini);
- facilità di interrogazione per operazioni frequenti (es. articoli in giacenza, storico operazioni);
- scalabilità futura per l'integrazione di nuove funzionalità, come il tracciamento avanzato o la gestione multi-magazzino.

[1] EAN (European Article Number) è uno standard internazionale di codifica a barre, utilizzato per identificare in modo univoco i prodotti nei sistemi di vendita e distribuzione.

[2] SKU (Stock Keeping Unit) è un codice alfanumerico univoco usato per tracciare e gestire un prodotto all'interno di un sistema di inventario.

3.4 Tecnologie abilitanti

A completamento dell'architettura MVC adottata per lo sviluppo dell'applicazione, si riportano di seguito le principali tecnologie impiegate per ciascun componente del sistema [Figura 3.1]:

- View: realizzata utilizzando TypeScript, con il supporto del framework Angular per la gestione dell'interfaccia utente e della libreria Bootstrap per la progettazione responsive e l'aspetto grafico.
- Controller: implementato in Java, attraverso il framework Spring Boot, responsabile della logica di business e dell'esposizione delle REST API.
- Model: basato su un database relazionale PostgreSQL, con gestione dei dati tramite SQL, mappato nel backend attraverso l'utilizzo di JPA.

3.4.1 Focus sulla view

Angular

Angular è un framework open-source sviluppato da Google, progettato per la realizzazione di applicazioni web moderne e dinamiche. Si basa su un'architettura a componenti, dove ogni componente incapsula al proprio interno un template HTML (HyperText Markup Language), uno stile CSS (Cascading Style Sheets) e una logica scritta in TypeScript, un superset tipizzato di JavaScript. Questo approccio favorisce la modularità, la riusabilità del codice e la manutenibilità dell'applicazione nel tempo.

Nel contesto del progetto, Angular è stato utilizzato per sviluppare l'interfaccia utente lato frontend, destinata all'uso da parte degli operatori di magazzino tramite tablet. Grazie all'approccio SPA (Single-Page Application), è stato possibile garantire un'esperienza di navigazione fluida e reattiva: le schermate vengono aggiornate dinamicamente, senza richiedere il ricaricamento completo della pagina a ogni interazione.

Angular fornisce inoltre un sistema di routing interno, che consente di gestire la navigazione tra le diverse viste dell'applicazione (es. login, ordini nuovi, ordini cancellati), assegnando

a ciascuna un percorso URL (Uniform Resource Locator)[3]. Ciò ha permesso di strutturare l'interfaccia in modo chiaro e coerente, facilitando l'accesso rapido alle funzionalità principali del sistema.

Bootstrap

Bootstrap è un framework CSS open-source sviluppato da Twitter, ampiamente utilizzato per la creazione di interfacce web responsive e uniformi. Fornisce una vasta raccolta di componenti grafici predefiniti, stili personalizzabili e classi utility che semplificano la progettazione dell'interfaccia utente, garantendo compatibilità e adattabilità su diversi dispositivi e risoluzioni.

All'interno del progetto, Bootstrap è stato impiegato in combinazione con Angular per la realizzazione della parte visiva dell'applicazione, con particolare attenzione all'usabilità da tablet, il dispositivo principale previsto per l'utilizzo da parte degli operatori di magazzino. L'utilizzo delle classi responsive ha permesso di adattare in modo dinamico la disposizione degli elementi grafici, assicurando una corretta visualizzazione e interazione anche su schermi di dimensioni contenute.

Inoltre, grazie ai componenti già pronti messi a disposizione dal framework (come pulsanti, tabelle, form, card, modali), è stato possibile velocizzare lo sviluppo dell'interfaccia utente, mantenendo uno stile coerente, ordinato e facilmente navigabile. L'impiego di Bootstrap ha quindi contribuito a rendere l'applicazione intuitiva e immediata, migliorando l'esperienza d'uso per l'utente finale.

3.4.2 Focus sul controller

Spring Boot

Spring Boot è un framework open-source basato su Java, che semplifica la creazione di applicazioni stand-alone e pronte per l'ambiente di produzione. Fa parte dell'ecosistema Spring e nasce con l'obiettivo di ridurre la configurazione manuale necessaria allo sviluppo di appli-

[3]L'URL è l'indirizzo univoco che specifica la posizione di una risorsa sul web e il protocollo da utilizzare per accedervi specifico.

cazioni web complesse, offrendo un set di funzionalità pronte all'uso (auto-configurazione, embedded server, gestione delle dipendenze, ecc.).

Nel contesto del progetto, Spring Boot è stato utilizzato per la realizzazione del backend dell'applicazione, con il compito di ricevere le richieste provenienti dal frontend Angular, elaborarle e restituire le risposte attraverso endpoint RESTful. Questi endpoint rappresentano la parte centrale del livello Controller nell'architettura MVC adottata, e sono responsabili della logica di coordinamento tra l'interfaccia utente, il sistema di autenticazione e il modello dati.

Spring Boot ha permesso di organizzare il codice backend in maniera modulare e scalabile, strutturando l'applicazione secondo un pattern ben definito (Controller $\rightarrow$ Service $\rightarrow$ Repository). Inoltre, grazie al supporto integrato per il server Tomcat embedded, è stato possibile eseguire e testare l'intera applicazione in modo semplice, senza la necessità di configurazioni esterne complesse.

Spring Data JPA

Spring Data JPA è un modulo del framework Spring che semplifica l'interazione tra il livello applicativo e il database, fornendo un'astrazione ad alto livello sopra la JPA. Grazie a questa tecnologia, è possibile eseguire operazioni CRUD e query complesse senza scrivere codice SQL esplicito, sfruttando l'uso di interfacce e convenzioni di naming.

Nel progetto sviluppato, Spring Data JPA è stato utilizzato per la definizione e la gestione del livello Model dell'architettura, rendendo possibile la mappatura automatica tra le entità Java e le tabelle presenti nel database relazionale PostgreSQL.

Attraverso l'annotazione delle classi (`@Entity`) e dei campi (`@Id`, `@Column`, ecc.), ogni entità è stata associata alla rispettiva tabella, garantendo una perfetta integrazione tra la logica del backend e la struttura del database. I repository definiti mediante interfacce (`JpaRepository`) hanno permesso di eseguire interrogazioni personalizzate e operazioni standard senza la necessità di implementare manualmente le logiche di accesso ai dati.

L'impiego di Spring Data JPA ha quindi reso il codice più conciso, modulare e facilmente manutenibile, facilitando allo stesso tempo la coerenza tra il dominio applicativo e il modello dati.

Spring Secuirity

Spring Security è un framework dell'ecosistema Spring progettato per gestire in modo completo e configurabile i meccanismi di autenticazione e autorizzazione all'interno delle applicazioni Java. Integra funzionalità avanzate per il controllo degli accessi, la protezione delle risorse e la gestione delle sessioni, offrendo al contempo un alto livello di personalizzazione.

Nel progetto realizzato, Spring Security è stato impiegato per implementare il sistema di autenticazione tramite nome utente e password e la generazione di token JWT. Nello specifico, al momento dell'accesso, l'utente inserisce le infomrazioni che vengono verificate rispetto ai valori associati nel database. In caso di esito positivo, il sistema genera un token JWT, che viene restituito al client e allegato alle successive richieste HTTP per autorizzare l'accesso alle funzionalità protette dell'applicazione.

Il token JWT consente una gestione stateless della sessione utente, poiché le informazioni di autenticazione sono contenute direttamente nel token stesso, evitando la necessità di mantenere uno stato lato server. Questo approccio è particolarmente adatto ad ambienti web distribuiti, come nel caso dell'applicazione in oggetto, in cui l'interfaccia frontend e il backend comunicano tramite REST API.

Spring Security ha inoltre permesso di definire in modo chiaro quali endpoint REST devono essere protetti e quali accessibili liberamente, garantendo così un livello adeguato di sicurezza e conformità ai principi di protezione dei dati. La configurazione personalizzata ha consentito di integrare il framework con la logica applicativa esistente, mantenendo la flessibilità e la coerenza dell'architettura.

3.4.3 Focus sul model

Per la gestione della persistenza dei dati è stato adottato un database relazionale PostgreSQL, scelto per la sua affidabilità, solidità e pieno supporto allo standard SQL. Tale tecnologia si è dimostrata particolarmente adatta a contesti applicativi come quello del magazzino, dove è necessario gestire in modo strutturato grandi quantità di dati e garantire la coerenza e l'integrità delle operazioni mediante l'uso di transazioni sicure.

All'interno dell'architettura MVC, il Model rappresenta le entità del dominio applicativo (come utenti, articoli, ordini e operazioni di magazzino), le cui istanze vengono mappate sulle tabelle del database attraverso l'utilizzo di Spring Data JPA. In questo modo è stato possibile modellare efficacemente le relazioni tra le diverse entità e garantire una perfetta integrazione tra la logica applicativa e la struttura del database.

Per la gestione e l'ispezione del database in fase di sviluppo è stato utilizzato DBeaver, un client grafico multi-piattaforma che ha facilitato la visualizzazione dei dati, l'esecuzione delle query e il monitoraggio delle operazioni di scrittura e lettura. L'uso di questo strumento si è rivelato particolarmente utile per il debugging e la verifica del corretto funzionamento delle funzionalità di backend durante la fase di test.

3.5 Mapping delle tecnologie abilitanti

Alla luce delle considerazioni esposte nei paragrafi precedenti, è possibile osservare in maniera sintetica la relazione tra il pattern architetturale MVC adottato e le tecnologie impiegate per la realizzazione delle rispettive componenti. La figura seguente mostra come ciascun livello dell'architettura sia stato supportato da strumenti e framework specifici, contribuendo alla costruzione di un sistema coeso, scalabile e manutenibile.

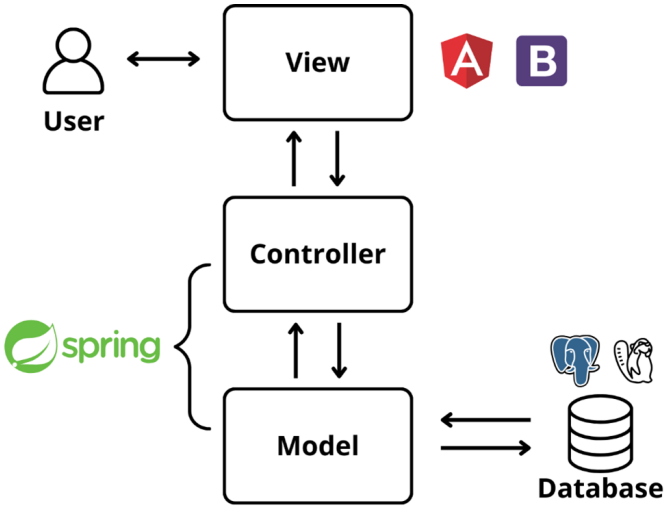


Figura 3.1: Associazione tra pattern MVC e tecnologie utilizzate nel progetto. Angular e Bootstrap per la View; Spring Boot, Spring Data JPA e Spring Security per Controller e Model; PostgreSQL come sistema di persistenza.

3.6 Implementazione

In questa sezione vengono presentati alcuni frammenti di codice significativi, selezionati allo scopo di illustrare nel dettaglio le principali scelte implementative effettuate durante lo sviluppo del prototipo. I blocchi di codice mostrati evidenziano il funzionamento delle componenti centrali dell'applicazione, come le operazioni sul magazzino e l'interazione con il database.

L'obiettivo di questa analisi è quello di fornire una visione concreta di come le tecnologie descritte nei paragrafi precedenti siano state applicate nella pratica, mostrando l'integrazione tra frontend, backend e livello di persistenza dei dati secondo l'architettura progettata.

3.6.1 Resoconto ordini

All'interno di questo paragrafo viene analizzata una delle funzionalità principali della sezione del gestionale sviluppata: la consultazione e gestione degli ordini presenti in magazzino. Questa componente rappresenta il punto di accesso per l'operatore, che ha la necessità di monitorare rapidamente lo stato degli ordini, accedere ai relativi dettagli, visualizzare i prodotti associati e interagire attraverso specifiche azioni (come accettazione, rifiuto o conferma della spedizione).

La funzionalità è stata progettata per essere modulare e scalabile, con particolare attenzione all'esperienza utente, supportando il caricamento asincrono dei dati, la paginazione dinamica e la possibilità di filtrare gli ordini per magazzino, stato o chiave testuale. Nei paragrafi successivi verranno analizzate nel dettaglio le principali componenti coinvolte, sia lato frontend che backend, evidenziando le interazioni tra le diverse tecnologie che compongono lo stack.

Logica applicativa – Backend

Il frammento di codice mostrato [Figura 3.2] rappresenta un esempio pratico di endpoint REST implementato nel backend tramite Spring Boot. La classe `OrdersController` è annotata con `@RestController`, indicando che essa ha il compito di esporre una o più interfacce RESTful accessibili da client esterni, in particolare dal frontend Angular. Inoltre, l'annota-

zione `@RequestMapping("/api/orders")` definisce il prefisso comune a tutti gli endpoint dichiarati all'interno della classe.

In questo caso, il metodo `getOrders()` è mappato sull'endpoint `/all` tramite l'annotazione `@GetMapping`, e ha il compito di restituire al frontend una lista di ordini. Tali ordini possono essere filtrati dinamicamente sulla base di alcuni parametri opzionali: il codice del magazzino (`warehouse`), una chiave di ricerca (`searchKey`) e lo stato dell'ordine (`status`).

Un ulteriore aspetto rilevante è la presenza della paginazione, implementata tramite i parametri `page` e `size`, che consente di suddividere i risultati in blocchi gestibili. Questo approccio migliora le prestazioni del sistema e garantisce una migliore esperienza utente, specialmente in scenari in cui il numero di ordini gestiti è elevato. I parametri `sortBy` e `direction` permettono inoltre di specificare l'ordinamento dei risultati.

Il metodo chiama quindi il livello service (`ordersService.getOrders(...)`), delegando ad esso la logica di business necessaria per elaborare la richiesta. L'uso di una struttura `PaginatedDtoResponse` permette di incapsulare gli ordini in una risposta formattata, pronta per essere consumata lato client.

```java
@RestController
@RequestMapping("/api/orders")
@RequiredArgsConstructor
public class OrdersController {

    private final OrdersService ordersService;

    @GetMapping("/all")  Antonio Romano
    public PaginatedDtoResponse<Orders, OrdersDtoResponse> getOrders(
            @RequestParam(value = "warehouse", required = false) String warehouseCode,
            @RequestParam(value = "searchKey", required = false) String searchKey,
            @RequestParam(value = "status", required = false) OrderStatusEnum orderStatusEnum,
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "id") String sortBy,
            @RequestParam(defaultValue = "desc") String direction) {
        return ordersService.getOrders(warehouseCode, searchKey, orderStatusEnum, page, size, sortBy, direction);
    }
```

Figura 3.2: Frammento di codice del controller dedicato al recupero degli ordini, con parametri dinamici e paginazione.

Il frammento di codice riportato [Figura 3.3] [Figura 3.4] rappresenta l'implementazione della logica di business nel metodo `getOrders` del Service, invocato dal controller REST mostrato in precedenza.

Una delle caratteristiche principali di questa implementazione è l'uso della Specification API di Spring Data JPA, che consente di costruire dinamicamente query complesse in base ai parametri ricevuti in input. In particolare, la variabile specification viene arricchita in modo incrementale sulla base dei filtri forniti: codice magazzino (`warehouseCode`), chiave di ricerca (`searchKey`) e stato dell'ordine (`orderStatusEnum`).

Prima di applicare il filtro sul magazzino, viene verificato che l'utente attualmente autenticato abbia effettivamente accesso al magazzino specificato, tramite una chiamata al metodo `authUtilsService.getWarehouseByAuthUser()`. Se il magazzino non è autorizzato, viene sollevata un'eccezione di tipo `GenericBadRequestException`. Inoltre, viene applicato un filtro che limita la visibilità degli ordini solo ai magazzini collegati all'utente.

Per quanto riguarda la ricerca testuale, il sistema permette di filtrare gli ordini attraverso diversi campi come codice dell'articolo, SKU e descrizione. Viene applicato un filtro di tipo `LIKE` con wildcard (%) per intercettare qualsiasi occorrenza parziale del termine.

Dopo aver costruito la Specification completa, viene eseguita la query paginata tramite `ordersRepository.findAll(...)`, che restituisce una pagina di risultati. Infine, gli ordini recuperati vengono convertiti in oggetti DTO (`OrdersDtoResponse`) e inseriti in un contenitore paginato da restituire al controller.

```
public PaginatedDtoResponse<Orders, OrdersDtoResponse> getOrders(String warehouseCode, String searchKey, OrderStatusEnum orderStatusEnum,   1 usage
                                                                 int page, int size, String sortBy, String direction) {
    Sort.Direction sortDirection = direction.equalsIgnoreCase( anotherString: "desc") ? Sort.Direction.DESC : Sort.Direction.ASC;
    Pageable pageable = PageRequest.of(page, size, Sort.by(sortDirection, sortBy));
    Specification<Orders> specification = ( Root<Orders> root,  CriteriaQuery<?> query,  CriteriaBuilder criteriaBuilder) -> criteriaBuilder.conjunction();

    if (warehouseCode != null && !warehouseCode.isBlank()) {
        Warehouse warehouseFilter = warehouseRepository.findByCode(warehouseCode).orElseThrow(() -> new GenericBadRequestException(warehouseCode + " not found"));
        Set<Warehouse> authWarehouses = authUtilsService.getWarehouseByAuthUser();
        if (!authWarehouses.contains(warehouseFilter))
            throw new GenericBadRequestException(warehouseCode + " is unknown");
        else {
            specification = specification.and(( Root<Orders> root,  CriteriaQuery<?> query,  CriteriaBuilder criteriaBuilder) ->
                    criteriaBuilder.equal(root.get("warehouse"), warehouseFilter));
        }
    } else {
        specification = specification.and(( Root<Orders> root,  CriteriaQuery<?> query,  CriteriaBuilder criteriaBuilder) ->
                root.get("warehouse").in(authUtilsService.getWarehouseByAuthUser()));
    }

    if (searchKey != null && !searchKey.isBlank()) {
        specification = specification.and(( Root<Orders> root,  CriteriaQuery<?> query,  CriteriaBuilder criteriaBuilder) -> criteriaBuilder.or(
                criteriaBuilder.like(root.get("code"),  pattern: "%" + searchKey + "%"),
                criteriaBuilder.like(root.get("articleCode"),  pattern: "%" + searchKey + "%"),
                criteriaBuilder.like(root.get("articleSKU"),  pattern: "%" + searchKey + "%"),
                criteriaBuilder.like(root.get("articleDescription"),  pattern: "%" + searchKey + "%")
        ));
    }
```

Figura 3.3: Parte 1: logica di costruzione dinamica della Specification per il recupero degli ordini in base ai parametri ricevuti.

```
if (orderStatusEnum != null) {
    specification = specification.and(( Root<Orders> root,  CriteriaQuery<?> query,  CriteriaBuilder criteriaBuilder) ->
            criteriaBuilder.equal(root.get("status"), orderStatusEnum));
}

Page<Orders> ordersPage = ordersRepository.findAll(specification, pageable);

List<OrdersDtoResponse> ordersDtoList = ordersPage.getContent().stream() Stream<Orders>
        .map(OrdersUtils::ordersToOrdersDTOResponse) Stream<OrdersDtoResponse>
        .toList();

return new PaginatedDtoResponse<>(ordersPage, ordersDtoList);
```

Figura 3.4: Parte 2: esecuzione della query con filtro e paginazione, conversione degli ordini in oggetti DTO e creazione della risposta.

Per la gestione del livello di persistenza, l'applicazione si affida all'uso di repository definiti come interfacce, in linea con le convenzioni di Spring Data JPA. Questo approccio permette di delegare gran parte del lavoro relativo all'accesso ai dati direttamente al framework, riducendo il codice boilerplate e garantendo una maggiore manutenibilità.

In particolare, la classe `OrdersRepository` [Figura 3.5] estende sia `JpaRepository`, che fornisce i metodi CRUD di base, sia `JpaSpecificationExecutor`, che abilita l'esecuzione di query dinamiche costruite tramite la Specification API. In questo caso, è stato sfruttato il metodo predefinito `findAll(Specification, Pageable)`, che consente di recuperare in modo semplice e conciso tutti gli ordini corrispondenti ai criteri specificati, con supporto integrato alla paginazione.

Per quanto riguarda i magazzini, l'interfaccia `WarehouseRepository` [Figura 3.6] estende anch'essa `JpaRepository` e definisce un metodo personalizzato `findByCode(String warehouseCode)`, che utilizza la convenzione di naming di Spring Data per generare automaticamente la query SQL necessaria alla ricerca del magazzino in base al suo codice identificativo.

```
@Repository  2 usages
public interface OrdersRepository extends JpaRepository<Orders, Long>, JpaSpecificationExecutor<Orders> {
```

Figura 3.5: Repository dedicato alla gestione degli ordini. Estende JpaRepository e JpaSpecificationExecutor, sfruttando il metodo predefinito `findAll` per eseguire query dinamiche con paginazione.

```java
@Repository  2 usages
public interface WarehouseRepository extends JpaRepository<Warehouse, Long> {
    Optional<Warehouse> findByCode(String warehouseCode);  1 usage  Antonio Romano
}
```

Figura 3.6: Repository dedicato all'entità `warehouse`, con definizione di un metodo personalizzato per la ricerca tramite codice.

Interfaccia utente – Frontend

Il recupero degli ordini dal backend avviene attraverso un insieme ben coordinato di metodi TypeScript presenti nel componente Angular della dashboard. Il comportamento dell'interfaccia è altamente dinamico e reattivo, grazie all'uso combinato della paginazione, del filtro testuale e della chiamata HTTP asincrona al servizio backend.

Il metodo `goToPage(page: number)` [Figura 3.8] si occupa di gestire la logica di paginazione. Verifica che il numero di pagina richiesto rientri nei limiti delle pagine disponibili (`0 <= page < totalPages`) e, in caso positivo, richiama `loadOrders(...)` passando come parametro la nuova pagina da caricare.

```html
<div class="flex-grow-0">
  <nav aria-label="Paginazione ordini" class="me-2">
    <ul class="pagination pagination-sm">

      <!-- Pulsante "Precedente" -->
      <li class="page-item" [class.disabled]="orders.first">
        <a class="page-link" (click)="goToPage( page: currentPage - 1)" aria-label="Previous">
          <span aria-hidden="true">«</span>
        </a>
      </li>

      <!-- Pagine dinamiche (mostra solo un range di 5 pagine) -->
      <li class="page-item">
        <a class="page-link" (click)="goToPage(startPage)">First</a>
      </li>

      <!-- Pagine dinamiche (mostra solo un range di 5 pagine) -->
      <li *ngFor="let i of [].constructor(endPage - startPage + 1); let index = index"
          class="page-item" [class.active]="(startPage + index) === currentPage">
        <a class="page-link" (click)="goToPage( page: startPage + index)">{{ startPage + index + 1 }}</a>
      </li>
```

Figura 3.7: Sezione della paginazione dinamica lato frontend. I pulsanti consentono all'utente di navigare tra le pagine disponibili e invocano il metodo `goToPage()` per il caricamento degli ordini relativi alla pagina selezionata.

```
goToPage(page: number): void {  Show usages
  if (page >= 0 && page < this.orders.totalPages) {
    this.loadOrders(page);
  }
}
```

Figura 3.8: Metodo `goToPage(page)` implementato nel componente Angular. Verifica che il numero della pagina rientri nei limiti consentiti e richiama il metodo `loadOrders(page)` per il recupero dei dati corrispondenti dal backend.

Il metodo `loadOrders(...)` rappresenta il cuore del recupero dati. Qui vengono costruiti dinamicamente gli HttpParams, che corrispondono esattamente ai parametri supportati dal controller REST `/api/orders/all`, analizzato in precedenza:

- `status`: stato dell'ordine
- `page`: numero della pagina corrente
- `size`: numero di ordini per pagina
- `sortBy`/`sortDirection`: ordinamento
- `warehouse`: filtro per magazzino selezionato
- `searchKey`: stringa di ricerca testuale

```
loadOrders(page: number): void {  Show usages
  let httpParams: HttpParams = new HttpParams()
    .set('status', this.statusFilter)
    .set('page', page)
    .set('size', 20)
    .set('sortBy', 'id')
    .set('sortDirection', 'DESC');
  if (this.currentWarehouse !== 'ALL')
    httpParams = httpParams.set('warehouse', this.currentWarehouse)
  if (this.searchKeyValue)
    httpParams = httpParams.set('searchKey', this.searchKeyValue)
```

Figura 3.9: Metodo `loadOrders(page)` utilizzato per il recupero degli ordini dal backend. I parametri vengono assemblati dinamicamente sulla base dello stato del filtro, del magazzino e della chiave di ricerca. La chiamata HTTP è asincrona e restituisce un oggetto contenente gli ordini e il numero di pagina attiva.

Una volta impostati i parametri, viene invocato `ordersService.getOrders` [Figura 3.10], che effettua la chiamata HTTP [Figura 3.11] vera e propria. Il risultato (`data`) viene salvato in `this.orders`, aggiornando anche `this.currentPage` e chiamando `updatePageRange()` per aggiornare la navigazione.

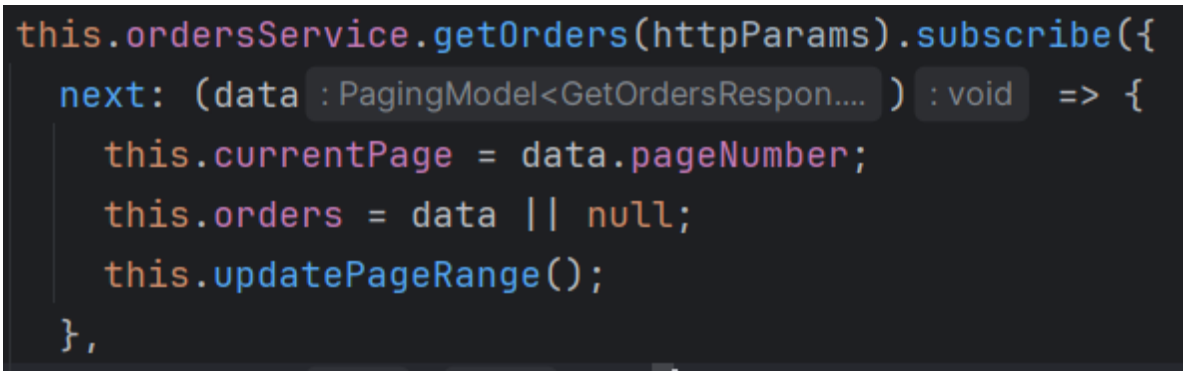


Figura 3.10: Blocco di sottoscrizione alla chiamata asincrona per il recupero degli ordini. I dati restituiti dal backend vengono assegnati alle variabili locali e viene aggiornato il range di paginazione.

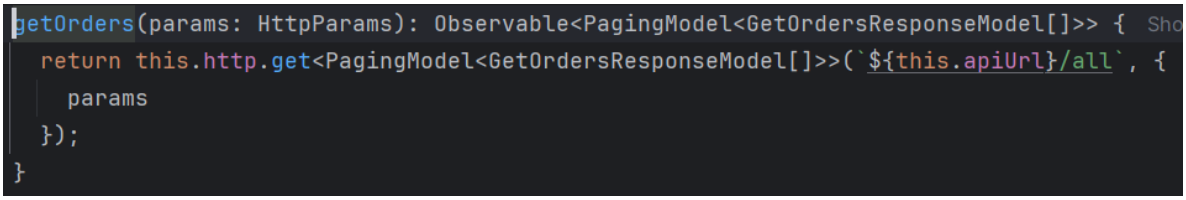


Figura 3.11: Metodo del service frontend che effettua la chiamata HTTP all'endpoint `/all` per il recupero degli ordini.

Un ulteriore aspetto che arricchisce il funzionamento del sistema di caricamento ordini è rappresentato dalla possibilità di effettuare una ricerca testuale. Quando l'utente inserisce un termine all'interno del campo di input e preme il tasto invio, il metodo `searchKey(...)` [Figura 3.13] intercetta l'evento, aggiorna il valore di ricerca (`searchKeyValue`) e richiama `loadOrders(...)`. Questo consente di filtrare dinamicamente gli ordini sulla base del contenuto testuale inserito, senza alterare la logica generale di caricamento, ma estendendone il comportamento in maniera trasparente ed efficace.

```
<div class="flex-grow-0 vw-100">
  <input type="text" class="form-control form-control-sm" placeholder="Inserisci il testo da cercare"
         (keyup.enter)="searchKey($event)"/>
</div>
```

Figura 3.12: Campo di input per la ricerca testuale. Quando l'utente preme il tasto Invio, viene invocato il metodo `searchKey($event)` per aggiornare i risultati mostrati.

```
searchKey($event: any) {  Show usages
  if ($event.target.value && $event.target.value.trim().length > 0) {
    this.searchKeyValue = $event.target.value;
  } else {
    this.searchKeyValue = null;
  }
  this.loadOrders(this.currentPage);
}
```

Figura 3.13: Metodo `searchKey(...)` che aggiorna il valore della chiave di ricerca e richiama il caricamento degli ordini filtrati. Il comportamento è condizionato dalla presenza o meno di un valore nel campo input.

3.6.2 Caricamento file per ottenere l'etichetta

Un'ulteriore funzionalità centrale nel sistema implementato riguarda l'upload di un file contenente la ricevuta di pagamento da parte dell'operatore. Questo processo, una volta completato con successo, attiva la generazione dell'etichetta di spedizione associata all'ordine selezionato.

Logica applicativa – Backend

Il metodo `sentLabel(...)` [Figura 3.14] è esposto tramite una richiesta HTTP POST all'endpoint `/label`, ed è definito all'interno del controller. Il parametro in ingresso è un oggetto DTO (`ShippingLabelOrderDtoRequest`) che viene gestito automaticamente dal framework Spring grazie all'uso combinato delle annotazioni `@Valid` e `@ModelAttribute`:

- `@ModelAttribute` si occupa di popolare l'oggetto DTO con i dati provenienti dalla richiesta HTTP, associando automaticamente i campi del form (o dei parametri) ai

campi del DTO. Questo consente al controller di ricevere un oggetto già costruito e pronto per essere elaborato.

- `@Valid` attiva il meccanismo di validazione dei dati sul DTO ricevuto, sulla base delle annotazioni presenti all'interno della classe (`@NotNull`). In caso di violazione dei vincoli, viene generata automaticamente una risposta di errore (con status code `400`), evitando l'esecuzione del metodo.

L'uso congiunto di queste due annotazioni permette di semplificare la gestione degli input, migliorare la sicurezza dell'applicazione e ridurre la necessità di controlli manuali all'interno del metodo.

Il controller si limita a delegare la logica di elaborazione al livello di servizio, mantenendo così una separazione netta delle responsabilità.

```
@PostMapping("/label")
public void sentLabel(@Valid @ModelAttribute ShippingLabelOrderDtoRequest shippingLabelOrderDtoRequest) {
    ordersService.getShippingLabels(shippingLabelOrderDtoRequest);
}
```

Figura 3.14: Metodo del controller dedicato alla ricezione della ricevuta di pagamento. Inoltra il DTO al service per l'elaborazione dell'etichetta.

Il metodo `getShippingLabels(...)` [Figura 3.15] implementa la logica completa per gestire questa operazione. Esso è annotato con `@Transactional`, a garanzia della consistenza delle operazioni sul database.

La procedura prevede i seguenti passaggi fondamentali:

- Validazione dell'input: se il file ricevuto è vuoto, viene lanciata un'eccezione personalizzata;
- Recupero dell'utente autenticato: utile per tracciare correttamente l'operazione;
- Recupero e verifica dell'ordine: se l'ordine non è presente o è già marcato come pronto all'invio, l'operazione viene interrotta;
- Upload del file su AWS: il file viene caricato su un bucket specifico tramite il servizio `awsBucketService`;
- Tracciamento dello stato: viene salvata un'entry all'interno della tabella di tracing dedicata alla cronologia degli ordini, utile a registrare in modo puntuale ogni varia-

zione di stato, insieme all'utente che l'ha eseguita e al relativo `timestamp` tramite `ordersStatusHistoryRepository`;

- Aggiornamento dello stato dell'ordine: lo stato viene aggiornato a `READY_TO_SEND` e salvato nel database;
- Risposta HTTP: viene restituito un messaggio di conferma.

Questo approccio consente di automatizzare un'operazione fondamentale per la preparazione alla spedizione, riducendo gli errori e migliorando l'efficienza operativa.

```
@Transactional  1 usage
public void getShippingLabels(ShippingLabelOrderDtoRequest shippingLabelOrderDtoRequest) {
    if (shippingLabelOrderDtoRequest.paymentReceipt().getSize() == 0) {
        throw new GenericBadRequestException("File is empty");
    }

    User user = authUtilsService.getAuthenticatedUser( useCache: true);

    Orders order = ordersRepository.findById(shippingLabelOrderDtoRequest.orderId())
            .orElseThrow(() -> new OrderNotFoundException("Orders not found"));

    if (OrderStatusEnum.READY_TO_SEND.equals(order.getStatus())) {
        throw new GenericBadRequestException("Lo stato è già impostato su "
                + order.getStatus());
    }

    String bucketName = shippingLabelOrderDtoRequest.bucketName();
    MultipartFile file = shippingLabelOrderDtoRequest.paymentReceipt();
    List<String> listErrors = awsBucketService.putObject(bucketName, file);

    ordersStatusHistoryRepository.save(OrdersUtils.traceStatusHistory(order, OrderStatusEnum.READY_TO_SEND, user));

    order.setStatus(OrderStatusEnum.READY_TO_SEND);
    ordersRepository.save(order);

    ResponseEntity.ok( body: "Pronto per l'invio!");
```

Figura 3.15: Logica del service che si occupa della validazione, upload della ricevuta, tracciamento dello stato e aggiornamento dell'ordine.

Il metodo `putObject(...)` [Figura 3.16] [Figura 3.17] è responsabile del caricamento di un file all'interno di un bucket AWS S3. È uno dei punti chiave dell'operazione di generazione dell'etichetta di spedizione, in quanto consente di archiviare in modo sicuro e centralizzato la ricevuta di pagamento inviata dall'operatore.

Questo metodo riceve in input:

- una stringa `key`, che rappresenta il percorso univoco del file nel bucket;

- un oggetto `MultipartFile` che incapsula il file da caricare.

L'operazione si articola in più fasi:

- Costruzione della richiesta: tramite il `builder` `PutObjectRequest`, vengono impostati il bucket di destinazione e la chiave (`path`) del file;
- Upload del file: il client AWS (`s3Client`) utilizza un `RequestBody` costruito a partire dallo stream del file;
- Gestione degli esiti:
  - in caso di successo, viene registrato un log di conferma;
  - in caso di errore, viene intercettata e gestita l'eccezione specifica:
    * `IOException` per errori di lettura del file;
    * `S3Exception` per problemi con il servizio AWS (es. permessi, connessioni, stato del bucket);
    * `SdkClientException` per errori generici di comunicazione con il client AWS.

```
public List<String> putObject(String key, MultipartFile file) {  1 usage
    List<String> errorList = new ArrayList<>();
    PutObjectRequest request = PutObjectRequest.builder().bucket(bucketName).key(key).build();
    log.info("Starting upload key {} on AWS S3 bucket {}", key, bucketName);
    try {
        s3Client.putObject(request, RequestBody.fromInputStream(file.getInputStream(), file.getSize()));
        log.info("Successfully upload on AWS S3 bucket {}", bucketName);
    } catch (IOException e) {
        errorList.add("Errore nella lettura del file: " + e.getMessage());
        log.error("IOException durante l'upload su S3", e);
```

Figura 3.16: Parte iniziale del metodo `putObject`: viene costruita la richiesta di upload verso Amazon S3 e viene avviato il caricamento del file tramite l'utilizzo del client `s3Client`, sfruttando uno stream sull'oggetto ricevuto. In caso di esito positivo, viene loggato un messaggio di conferma.

In ogni blocco di `catch`, viene registrato un messaggio di errore dettagliato, utile per il monitoraggio e il debug, e l'eccezione viene rilanciata per interrompere il flusso. Gli eventuali messaggi di errore vengono accumulati in una lista (`errorList`) restituita dal metodo, in modo che il chiamante possa decidere come gestirli.

```
    } catch (S3Exception e) {
        errorList.add("Il contratto: " + key + " è andato in errore per il server aws, sul bucketName: " +
                bucketName + " NON INSERITO."
                + " Messaggio: " + e.awsErrorDetails().errorMessage()
                + " HTTP Status Code: " + e.awsErrorDetails().errorCode()
                + " AWS Error Code: " + e.awsErrorDetails().errorMessage());
        e.printStackTrace();
        throw e;
    } catch (SdkClientException e) {
        errorList.add("Failed to upload file to S3: " + e.getMessage());
        e.printStackTrace();
        throw e;
    }
    return errorList;
}
```

Figura 3.17: Sezione conclusiva del metodo `putObject`, dedicata alla gestione delle eccezioni. In caso di errori durante l'upload, viene popolata una lista di messaggi di errore esplicativi e viene lanciata l'eccezione corrispondente. I messaggi includono i dettagli provenienti dal servizio AWS.

CHAPTER

4 Il funzionamento del prototipo

Dopo aver analizzato la progettazione e l'implementazione della sezione del gestionale realizzata, in questo capitolo si intende descriverne il funzionamento effettivo, illustrando come l'utente possa interagire con l'interfaccia e con le funzionalità integrate.

L'obiettivo è mostrare, attraverso alcuni casi d'uso rappresentativi, in che modo il sistema supporti concretamente le operazioni quotidiane svolte dagli operatori di magazzino, dalla consultazione degli ordini alla gestione delle spedizioni. Le schermate e i comportamenti descritti fanno riferimento al prototipo sviluppato durante il tirocinio e pensato per offrire un'interazione semplice, rapida e sicura.

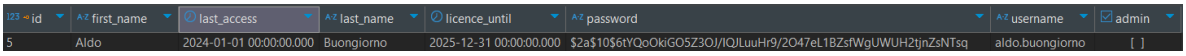
I paragrafi successivi accompagneranno il lettore attraverso una panoramica pratica dell'applicazione, evidenziando i punti chiave dell'esperienza utente e il legame diretto con le logiche implementate lato backend.

4.1 I dati

Prima di analizzare il comportamento del sistema nei casi d'uso concreti, è opportuno presentare una panoramica dei dati presenti nel database al momento dell'esecuzione del prototipo.

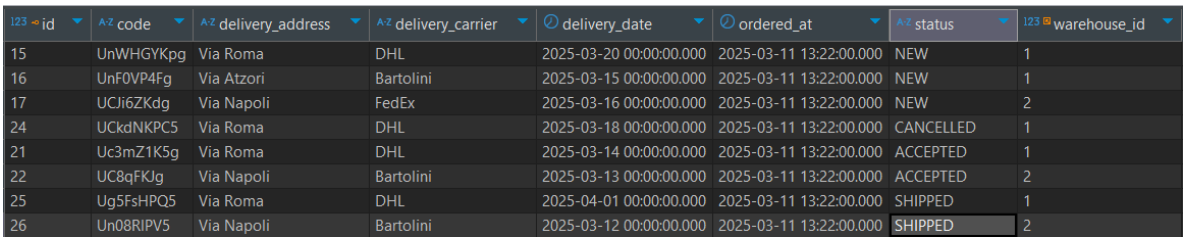
A tal fine, sono state popolate le tabelle del sistema con valori fittizi, utilizzati esclusivamente a scopo dimostrativo. Questi dati consentono di simulare realisticamente l'interazione dell'utente con l'applicazione, senza fare riferimento ad alcuna informazione sensibile o reale.

Le tabelle illustrate in questo paragrafo rappresentano entità centrali nel contesto operativo del gestionale, come ordini, articoli, magazzini e utenti, e costituiranno la base informativa a cui si farà riferimento nei paragrafi successivi.



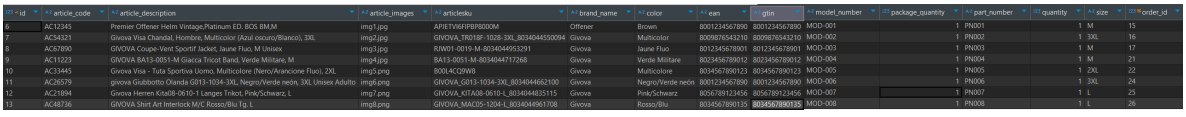
id	first_name	last_access	last_name	licence_until	password	username	admin
5	Aldo	2024-01-01 00:00:00.000	Buongiorno	2025-12-31 00:00:00.000	$2a$10$6tYQoOkiGO5Z3OJ/IQJLuuHr9/2O47eL1BZsfWgUWUH2tjnZsNTsq	aldo.buongiorno	[ ]

Figura 4.1: Contenuto dimostrativo della tabella `users`, contenente i dati di un utente abilitato all'accesso. Si noti che la password è salvata in forma crittografata tramite algoritmo `bcrypt`, utilizzato da Spring Security, a garanzia della sicurezza e dell'integrità delle credenziali memorizzate. Tutti i dati presenti sono fittizi e inseriti esclusivamente a scopo illustrativo.



id	code	delivery_address	delivery_carrier	delivery_date	ordered_at	status	warehouse_id
15	UnWHGYKpg	Via Roma	DHL	2025-03-20 00:00:00.000	2025-03-11 13:22:00.000	NEW	1
16	UnF0VP4Fg	Via Atzori	Bartolini	2025-03-15 00:00:00.000	2025-03-11 13:22:00.000	NEW	1
17	UCJi6ZKdg	Via Napoli	FedEx	2025-03-16 00:00:00.000	2025-03-11 13:22:00.000	NEW	2
24	UCkdNKPC5	Via Roma	DHL	2025-03-18 00:00:00.000	2025-03-11 13:22:00.000	CANCELLED	1
21	Uc3mZ1K5g	Via Roma	DHL	2025-03-14 00:00:00.000	2025-03-11 13:22:00.000	ACCEPTED	1
22	UC8qFKJg	Via Napoli	Bartolini	2025-03-13 00:00:00.000	2025-03-11 13:22:00.000	ACCEPTED	2
25	Ug5FsHPQ5	Via Roma	DHL	2025-04-01 00:00:00.000	2025-03-11 13:22:00.000	SHIPPED	1
26	Un08RIPV5	Via Napoli	Bartolini	2025-03-12 00:00:00.000	2025-03-11 13:22:00.000	SHIPPED	2

Figura 4.2: Contenuto dimostrativo della tabella `orders` prima dell'interazione dell'utente con il sistema. I dati mostrano gli ordini inizialmente presenti nel gestionale, ciascuno associato a un codice identificativo, a un indirizzo di consegna, al corriere e a un magazzino di riferimento. Lo stato dell'ordine (`status`) sarà oggetto di modifiche nelle simulazioni successive, al fine di illustrare il funzionamento del prototipo sviluppato.



id	article_code	article_description	article_images	articlesku	brand_name	color	ean	gtin	model_number	package_quantity	part_number	quantity	size	order_id
6	AC12345	Premier Offener Helm Vintage,Platinum ED. BOS BM,M	img1.jpg	APJETVI6FIPBP8000M	Offener	Brown	8001234567890	8001234567890	MOD-001	1	PN001	1	M	15
7	AC54321	Givova Visa Chandal, Hombre, Multicolor (Azul oscuro/Blanco), 3XL	img2.jpg	GIVOVA_TR018F-1028-3XL_8034044550094	Givova	Multicolor	8009876543210	8009876543210	MOD-002	1	PN002	1	3XL	16
8	AC67890	GIVOVA Coupe-Vent Sportif Jacket, Jaune Fluo, M Unisex	img3.jpg	RJW01-0019-M-8034044053291	Givova	Jaune Fluo	8012345678901	8012345678901	MOD-003	1	PN003	1	M	17
9	AC11223	GIVOVA BA13-0051-M Giacca Tricot Band, Verde Militare, M	img4.jpg	BA13-0051-M-8034044717268	Givova	Verde Militare	8023456789012	8023456789012	MOD-004	1	PN004	1	M	21
10	AC13445	Givova Visa - Tuta Sportiva Uomo, Multicolore (Nero/Arancione Fluo), 2XL	img5.png	B001/4C/20W8	Givova	Multicolore	8034567890123	8034567890123	MOD-005	1	PN005	1	2XL	22
11	AC26579	givova Giubbotto Olanda G013-1034-3XL, Negro/Verde neón, 3XL Unisex Adulto	img6.png	GIVOVA_G013-1034-3XL_8034044662100	Givova	Negro/Verde neón	8001234567890	8001234567890	MOD-006	1	PN006	1	3XL	24
12	AC21894	Givova Herren Kita08-0610-1 Langes Trikot, Pink/Schwarz, L	img7.png	GIVOVA_KITA08-0610-L_8034044835115	Givova	Pink/Schwarz	8056789123456	8056789123456	MOD-007	1	PN007	1	L	25
13	AC40736	GIVOVA Shirt Art Interlock M/C Rosso/Blu Tg.L	img8.png	GIVOVA_MAC05-1204-L_8034044961708	Givova	Rosso/Blu	8034567890135	8034567890135	MOD-008	1	PN008	1	L	26

Figura 4.3: Contenuto dimostrativo delle tabelle `order_items` e `orders`. I dati riportati, a scopo esemplificativo, rappresentano i dettagli degli articoli associati agli ordini presenti nel sistema. L'immagine mostra le informazioni anagrafiche del prodotto, come codice, descrizione, brand, codici a barre, SKU,quantità, taglia, riferimento all'ordine di appartenenza. Questo consente una tracciabilità completa del contenuto di ciascun ordine.

Infine, si segnala che la tabella dedicata allo storico degli stati degli ordini (`orders_status_history`) risulta inizialmente vuota, in quanto verrà popolata dinamicamente durante l'esecuzione delle operazioni analizzate nei paragrafi successivi. Analogamente, non vengono riportati i dati relativi alle tabelle di supporto al magazzino (come `warehouse` e `users_warehouses`), in quanto non direttamente coinvolti nelle funzionalità dimostrate. Tuttavia, tali informazioni

sono presenti e correttamente strutturate nel database, a garanzia dell'integrità, della coerenza e del corretto funzionamento complessivo del sistema.

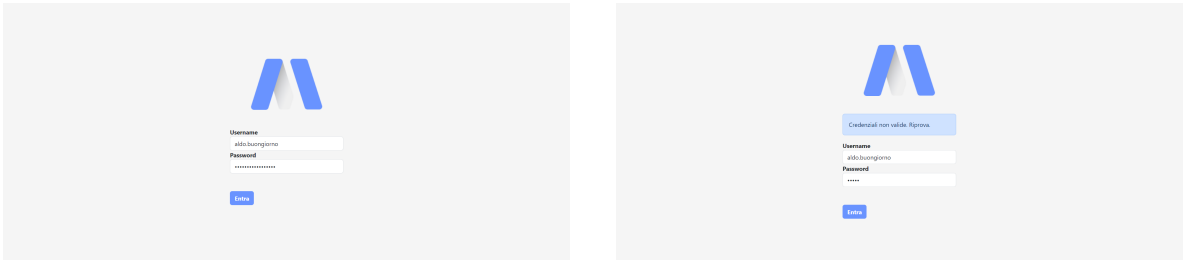
4.2 Scenario applicativo

Per illustrare il funzionamento del prototipo sviluppato, nei paragrafi seguenti verranno presentate alcune schermate dell'applicazione estratte da sessioni di utilizzo simulate. Le immagini, catturate da desktop, mostrano passo dopo passo l'interazione dell'utente con il sistema nelle principali funzionalità implementate, con lo scopo di rendere chiara la logica operativa alla base del gestionale.

4.2.1 Login

Il punto di accesso alla sezione del gestionale progettata è rappresentato dalla schermata di login [Figura 4.4]. In questa fase, l'utente deve autenticarsi inserendo le proprie credenziali, costituite da username e password, affinché gli venga concesso l'accesso al sistema. Tale procedura costituisce una prima importante misura di sicurezza e, nel contesto applicativo proposto, assume ulteriore rilevanza in quanto consente anche di associare le azioni svolte nel sistema a uno specifico operatore. Nel prototipo realizzato, una volta effettuato correttamente l'accesso, viene restituito un token JWT che consente l'autenticazione alle successive richieste.

Nel caso in cui l'utente inserisca delle credenziali non valide, il sistema restituisce un messaggio d'errore informativo, segnalando l'impossibilità di effettuare l'accesso. L'interfaccia mantiene comunque un aspetto coerente e chiaro, permettendo all'utente di ritentare immediatamente l'autenticazione. Questo tipo di feedback è fondamentale per l'usabilità dell'applicazione, poiché consente di guidare l'utente nella correzione dell'errore senza interrompere il flusso operativo.



(a) Accesso corretto (b) Accesso negato

Figura 4.4: Schermate del login dell'applicazione. A sinistra l'accesso corretto con credenziali valide, a destra il messaggio d'errore restituito in caso di autenticazione fallita.

4.2.2 Schermata principale

Dopo aver effettuato correttamente l'accesso, l'utente viene reindirizzato alla schermata principale dell'applicazione, nella quale viene presentata la sezione *Ordini Nuovi* [Figura 4.5]. In questa vista, il sistema mostra tutti gli ordini presenti nel database con stato attuale `NEW`, cioè non ancora presi in carico o elaborati.

Per ciascun ordine vengono visualizzate informazioni fondamentali come:

- Codice identificativo dell'ordine
- Numero di prodotti e quantità totale
- Magazzino di riferimento
- Dettagli del prodotto, quali descrizione, SKU, immagine e data di inserimento.

L'interfaccia è progettata per permettere all'operatore di prendere decisioni rapide, grazie alla presenza di due pulsanti ben distinti: *Accetta* e *Rifiuta*, che rappresentano rispettivamente l'avvio o il rifiuto della gestione di quell'ordine.

È inoltre presente un sistema di paginazione che consente una navigazione agevole tra i diversi gruppi di ordini. Oltre ai numeri di pagina, vengono mostrati anche i pulsanti:

- ≪First≫ per tornare rapidamente alla prima pagina;
- ≪Last≫ per accedere all'ultima;
- "≪" e "≫" per procedere alla precedente o successiva.

Questo sistema garantisce un'esperienza utente fluida anche in presenza di un elevato numero di ordini, evitando sovraccarichi visivi e migliorando l'usabilità.

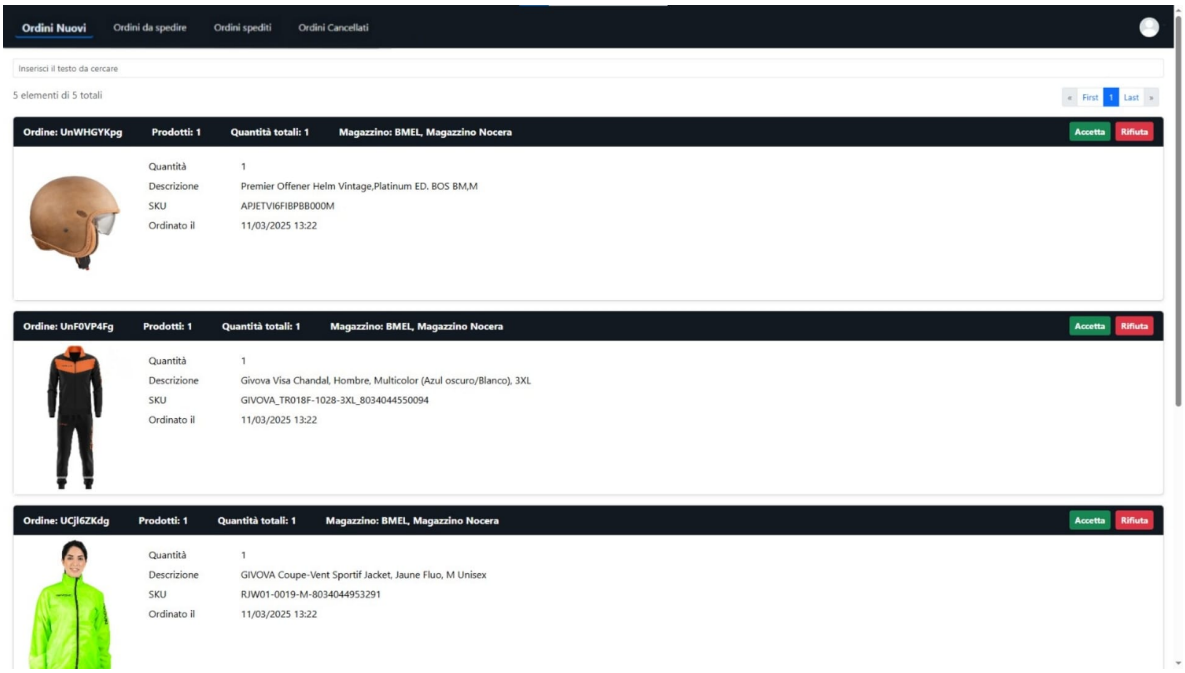


Figura 4.5: Pagina principale del prototipo dopo il login, in cui vengono elencati gli ordini presenti nel sistema con stato `NEW`, ovvero in attesa di essere elaborati. Ogni scheda mostra dettagli utili all'operatore, come il prodotto, la quantità e il magazzino di riferimento. La sezione include anche pulsanti di azione rapida (*Accetta* e *Rifiuta*) e un sistema di paginazione con controlli *First*, *Last*, $\ll$ e $\gg$ per facilitare la consultazione.

Per migliorare la consultazione degli ordini presenti nel sistema, è stata implementata una funzionalità di filtraggio testuale. L'utente può inserire liberamente una parola chiave (come ad esempio un codice ordine, uno SKU o una descrizione) nell'apposito campo di ricerca in alto. Il sistema provvede quindi ad aggiornare dinamicamente la lista degli ordini mostrati, restituendo esclusivamente quelli che contengono al loro interno la parola digitata.

Questo meccanismo incrementa notevolmente l'efficienza operativa, specialmente in scenari dove il numero di ordini da gestire è elevato. L'interfaccia rimane reattiva e facilmente navigabile anche a fronte di query più specifiche, garantendo un'esperienza fluida per l'utente.

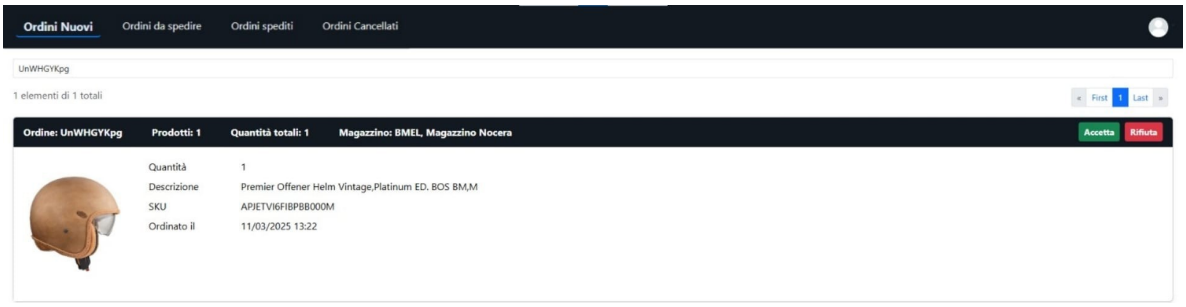
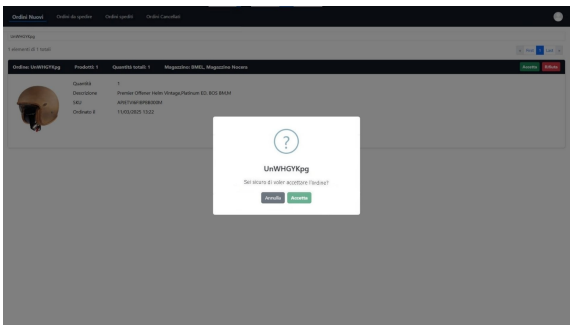


Figura 4.6: Esempio di utilizzo del campo di ricerca per il filtraggio dinamico degli ordini. Inserendo una chiave di ricerca, l'interfaccia aggiorna in tempo reale la visualizzazione, mostrando solo gli ordini corrispondenti al criterio inserito.

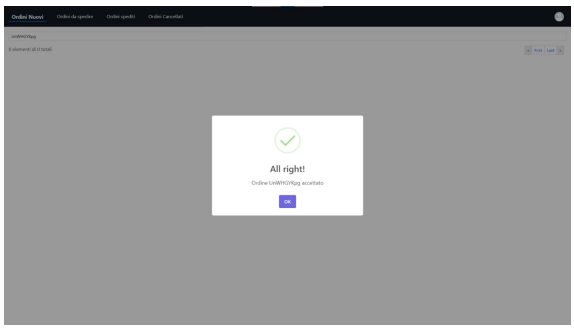
4.2.3 Accettazione ordine

Una volta identificato un ordine da processare, l'utente può cliccare sul pulsante *Accetta*, situato nella parte destra della card corrispondente all'ordine. Questa azione non genera effetti immediati, ma apre una modale di conferma in cui viene richiesto all'operatore di verificare e confermare l'intenzione di procedere. Solo dopo aver cliccato nuovamente su *Accetta* nel popup, l'ordine viene effettivamente accettato.

A seguito della conferma, l'interfaccia mostra una seconda modale che segnala l'esito positivo dell'operazione. L'ordine viene così rimosso dalla lista degli ordini nuovi e il suo stato aggiornato, risultando ora visibile nella sezione *Ordini da spedire*.



(a) Modale di conferma accettazione



(b) Modale di avvenuta accettazione

Figura 4.7: Fasi del processo di accettazione di un ordine. A sinistra la modale di conferma che previene operazioni accidentali, a destra la notifica che conferma l'avvenuto aggiornamento dello stato.

4.2.4 Ordine spedito

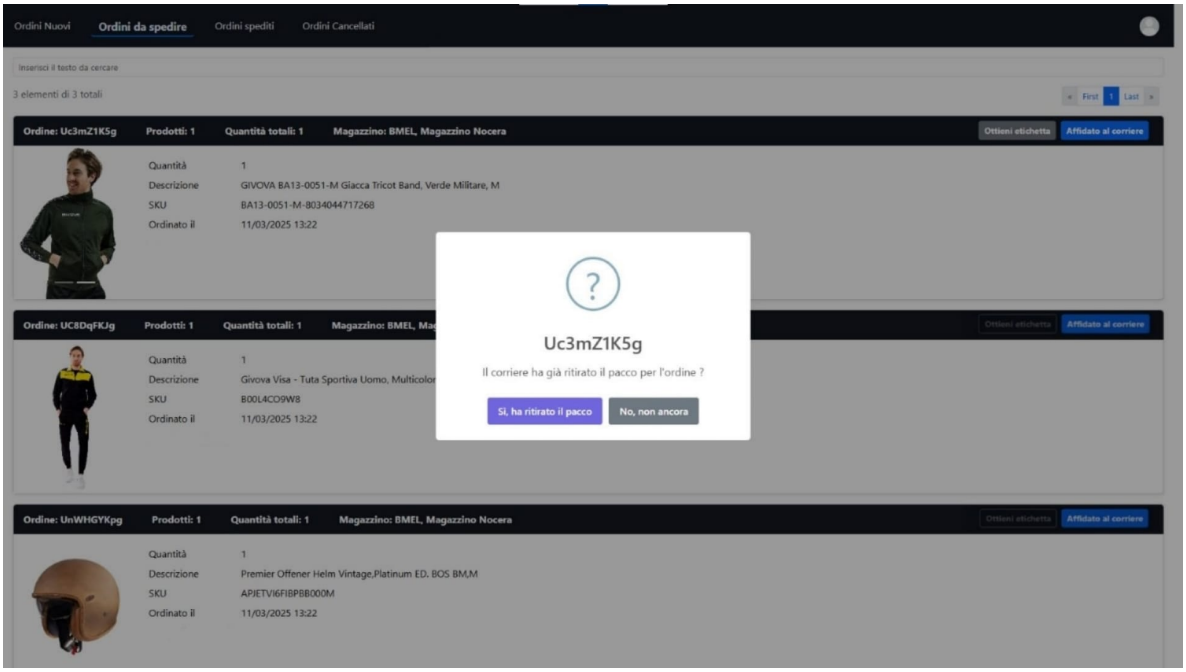


Figura 4.8: Modale di conferma per l'affidamento dell'ordine al corriere. L'operatore ha la possibilità di dichiarare se il pacco è stato effettivamente ritirato selezionando una delle due opzioni: *"Sì, ha ritirato il pacco"* o *"No, non ancora"*.

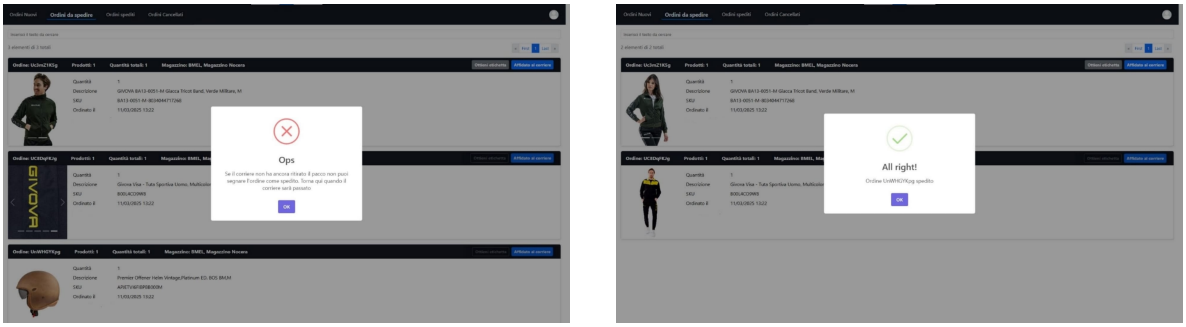
Una volta che un ordine è stato accettato, questo compare nella sezione *Ordini da spedire*, pronto per essere affidato al corriere incaricato della consegna. L'interfaccia mostra, accanto a ogni ordine, il pulsante *Affidato al corriere*, che consente all'operatore di registrare la presa in carico del pacco.

Alla pressione del pulsante, viene inizialmente visualizzata una modale di conferma [Figura 4.8] che chiede se il corriere abbia effettivamente già ritirato il pacco. L'utente può quindi scegliere se proseguire o annullare l'operazione [Figura 4.9].

Se si seleziona *No, non ancora*, viene mostrata una notifica di errore che impedisce la registrazione dell'operazione, invitando l'operatore a riprovare quando il pacco sarà stato effettivamente ritirato.

Al contrario, se l'operatore conferma il ritiro selezionando *Sì, ha ritirato il pacco*, il sistema registra correttamente la presa in carico da parte del corriere e mostra un messaggio di

conferma. A seguito di questa azione, l'ordine scompare dalla lista corrente, poiché non è più in attesa di spedizione.



(a) Affidamento fallito (b) Affidamento riuscito

Figura 4.9: Gestione dell'affidamento al corriere: a sinistra, il sistema blocca l'operazione se il ritiro non è ancora avvenuto; a destra, conferma dell'affidamento andato a buon fine.

4.2.5 Ordine rifiutato

In alcuni casi può rendersi necessario rifiutare un ordine. Per garantire la tracciabilità e la chiarezza delle motivazioni, il sistema prevede l'apertura di una modale dedicata alla conferma del rifiuto [Figura 4.10]. L'utente è invitato a specificare obbligatoriamente la motivazione della decisione, con un vincolo minimo di 10 caratteri.

Solo dopo aver compilato correttamente il campo testuale, sarà possibile confermare l'azione premendo il pulsante *Rifiuta*. A seguito di questa operazione, l'ordine verrà rimosso dalla lista degli ordini nuovi e inserito nella sezione *Ordini cancellati*, in cui saranno conservate anche le informazioni associate alla motivazione del rifiuto.

Questa misura, oltre a garantire la tracciabilità delle operazioni svolte dagli operatori, consente di raccogliere dati utili per analisi di mercato più accurate. Ad esempio, la frequente segnalazione di un prodotto non disponibile come motivo di rifiuto può indicare una domanda ricorrente non soddisfatta, suggerendo interventi mirati su scorte, fornitori o priorità logistiche.

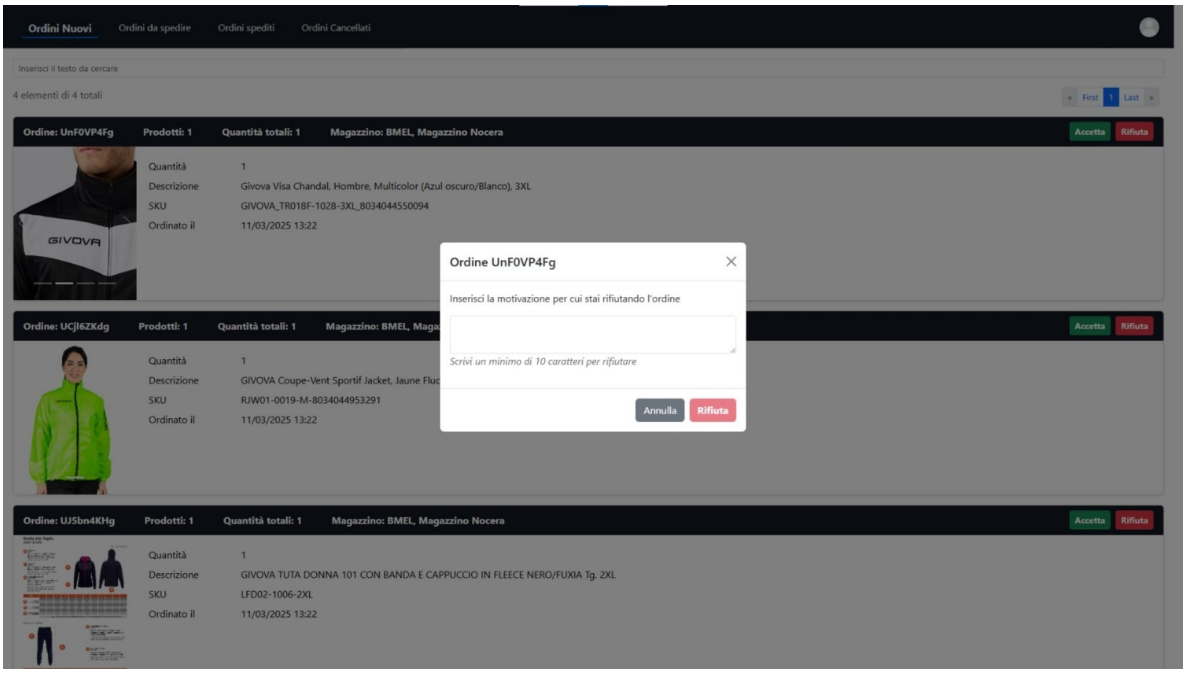


Figura 4.10: Modale per il rifiuto di un ordine. Il sistema richiede l'inserimento obbligatorio di una motivazione, vincolata a un minimo di 10 caratteri, prima di procedere all'eliminazione dell'ordine dalla lista corrente.

4.3 Test e riscontro degli utenti

Tutte le funzionalità implementate all'interno dell'applicazione sono state testate con esito positivo. I test si sono svolti sotto la supervisione del programmatore di Magusa e con il coinvolgimento diretto degli operatori di magazzino, principali utilizzatori del sistema. I feedback raccolti hanno confermato la bontà del lavoro svolto, evidenziando un'esperienza d'uso intuitiva, una buona reattività dell'interfaccia e un comportamento complessivamente affidabile del sistema nelle diverse condizioni operative.

4.4 Conclusioni e sviluppi futuri

La sezione del gestionale sviluppata rappresenta una prima versione funzionale del modulo dedicato alla gestione degli ordini e delle etichette di spedizione. L'applicazione, così com'è, copre efficacemente il flusso operativo principale previsto per i magazzinieri, offrendo un'in-

terfaccia intuitiva e strumenti utili alla gestione quotidiana delle attività. Tuttavia, il progetto presenta ampi margini di miglioramento e potenziamento.

Tra gli sviluppi futuri previsti, assume particolare rilevanza l'integrazione della comunicazione con il *tool master*, che rappresenta il nucleo centrale del sistema gestionale aziendale. Tale comunicazione è essenziale per abilitare funzionalità avanzate, tra cui il coordinamento automatico tra i vari componenti software e la sincronizzazione degli stati dell'ordine.

Inoltre, sebbene la logica per il download dei file dal bucket Amazon S3 sia già stata predisposta lato codice, tale funzionalità non è ancora attiva nell'ambiente attuale a causa di limitazioni sui permessi di accesso. Al momento è infatti possibile eseguire soltanto l'upload dei file. Sarà dunque necessario, in una fase successiva, testare e adeguare questa funzionalità, verificando il corretto scambio dei dati con il *tool master* e aggiornando le configurazioni di accesso al bucket S3.

In sintesi, il lavoro svolto costituisce una solida base su cui costruire ulteriori sviluppi, puntando a una soluzione sempre più completa, integrata ed efficiente a supporto delle attività logistiche aziendali.

Bibliografia

[1] Paolo Atzeni et al. *Basi di Dati*. 6ª ed. McGraw-Hill Education, 2023. ISBN: 9788838659584.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. 2ª ed. Addison-Wesley, 2004. ISBN: 9780321278654.

[3] T. Berners-Lee, R. Fielding e L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986, Internet Engineering Task Force. Standards Track. 2005. URL: `https://www.rfc-editor.org/rfc/rfc3986` (visitato il giorno 13/04/2025).

[4] Mike Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. RFC 9114, Internet Engineering Task Force. Standards Track. 2022. URL: `https://datatracker.ietf.org/doc/html/rfc9114` (visitato il giorno 12/04/2025).

[5] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259, Internet Engineering Task Force. Standards Track. 2017. URL: `https://www.rfc-editor.org/rfc/rfc8259` (visitato il giorno 13/04/2025).

[6] T. Bray et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. 2008. URL: `https://www.w3.org/TR/xml/` (visitato il giorno 13/04/2025).

[7] Claudio De Sio Cesari. *Manuale di Java 8. Programmazione orientata agli oggetti con Java standard*. 8ª ed. Hoepli, 2014. ISBN: 978-8820362911.

[8] Tsui Frank F., Bernal Barbara e Karam Orlando. *Essentials of Software Engineering*. 5ª ed. Jones & Bartlett Learning, 2022. ISBN: 9781284228991.

[9] Roy Thomas Fielding. ≪Architectural Styles and the Design of Network-based Software Architectures≫. Ph.D. Dissertation. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (visitato il giorno 12/04/2025).

[10] M. Jones, J. Bradley e N. Sakimura. *JSON Web Token (JWT)*. RFC 7519, Internet Engineering Task Force. Standards Track. 2015. URL: https://tools.ietf.org/html/rfc7519 (visitato il giorno 12/04/2025).

[11] Trygve Reenskaug. *Thing-Model-View-Editor — an Example from a Planning System*. Xerox PARC internal report. Original proposal of MVC pattern. 1979. URL: https://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html (visitato il giorno 13/04/2025).

Sitografia

[12] Wikipedia contributors. *Amazon S3*. URL: https://en.wikipedia.org/wiki/Amazon_S3 (visitato il giorno 25/03/2025).

[13] Microsoft. *TypeScript: JavaScript With Syntax for Types*. 2024. URL: https://www.typescriptlang.org/ (visitato il giorno 12/04/2025).

[14] Angular Team. *Angular Documentation*. 2024. URL: https://angular.io/docs (visitato il giorno 13/04/2025).

[15] Spring Team. *Spring Boot Documentation*. 2024. URL: https://spring.io/projects/spring-boot (visitato il giorno 13/04/2025).

[16] The Bootstrap Team. *Bootstrap – The Most Popular HTML, CSS, and JS Library*. 2024. URL: https://getbootstrap.com/ (visitato il giorno 12/04/2025).

Acronimi

DF Direct Fulfillment

AWS Amazon Web Service

UX User Experience

JWT JSON Web Token

RDBMS Sistema di Gestione di Basi di Dati Relazionali – *Relational DataBase Management System* –

Magusa Magusa S.r.l

MVC Model-View-Controller

REST API REpresentational State Transfer Application Programming Interface

REST REpresentational State Transfer

API Application Programming Interface

HTTP HyperText Transfer Protocol

JSON JavaScript Object Notation

XML Extensible Markup Language

CRUD Create,Read,Update,Delete

URI Uniform Resource Identifier

SQL Structured Query Language

DML Data Manipulation Language

DDL Data Definition Language

DQL Data Query Language

DCL Data Control Language

ACID Atomicity, Consistency, Isolation, Durability

XP Extreme Programming

JPA Java Persistence API

SPA Single-Page Application

HTML HyperText Markup Language

CSS Cascading Style Sheets

URL Uniform Resource Locator

DTO Data Transfer Object

SKU Stock Keeping Unit

EAN European Article Number