

Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate (DiSTA)
Corso di Laurea Triennale in Informatica



Programmazione e implementazione di un gestionale full-stack per il commercio elettronico multicanale

Relatore: Prof.ssa Ferrari Elena

Tesi di Laurea di
Mauri Andrea
Matricola 755130

Anno Accademico 2024-2025

Ringraziamenti

Desidero esprimere la mia sincera gratitudine alla Prof.**ssa Elena Ferrari**, relatrice di questa tesi, per la disponibilità, la fiducia e l'attenzione dedicata durante tutto il percorso.

Un ringraziamento speciale va all'**ing. Andrea Abba**, tutor aziendale, per avermi guidato con pazienza e competenza nello sviluppo del progetto all'interno dell'azienda. La sua esperienza è stata fondamentale per comprendere il contesto reale e trasformare un'idea in una soluzione concreta.

Ringrazio inoltre **l'azienda Abba Cavallini** per avermi dato l'opportunità di affrontare un progetto vero e stimolante, mettendomi alla prova in un ambiente professionale e collaborativo.

Un grazie profondo ai miei genitori, **Silvano e Damonte**, per il loro supporto costante, i sacrifici e la fiducia che mi hanno sempre dimostrato. Alla mia fidanzata, **Sofia**, che mi ha sostenuto nei momenti di stress e ha creduto in me anche quando io stesso esitavo.

Grazie ai miei compagni di università, **Luca, Paolo e Marco**, con cui ho condiviso fatiche, esami e soddisfazioni. E agli amici di sempre, **Mattia, Simone e Francesco**, per avermi tenuto connesso alla realtà, tra una pausa e l'altra.

Infine, un ringraziamento "non umano" ma doveroso va a **Stack Overflow, YouTube**, e anche a **ChatGPT**: senza i loro suggerimenti, spunti e bugfix silenziosi, alcune giornate di programmazione sarebbero finite molto peggio.

A tutti voi, grazie.

“Misurare i progressi della programmazione in base alle linee di codice è come misurare i progressi nella costruzione di aerei in base al peso.”

— *Bill Gates*

Indice

CAP. 1	12
1.1 - Globalizzazione e necessità per le piccole realtà alla vendita online	13
1.2 - Canali di vendita online, frazionamento del mercato	14
1.3 - Gestione di magazzini dislocati e canali di vendita eterogenei tramite un unico gestionale	15
1.4 - Perché sviluppare un sistema gestionale personalizzato e non usare soluzioni già esistenti	16
CAP. 2	18
2.1 - Vantaggi nell'utilizzare framework sia per back-end che il front-end	19
2.1.1 - Esempio “Prima e Dopo” l’adozione di un framework Back-end	20
2.1.2 - Esempio “Prima e Dopo” l’adozione di un framework Front-end	21
2.2 - Approccio moderno per la strutturazione database tramite ORM	21
2.2.1 - Esempio “Prima e Dopo” l’adozione di un ORM	23
2.3 - Django framework	23
2.4 - React Framework	24
2.5 - Restful API	25
2.5.1 - Esempio in Django Rest Framework	26
2.6 - GraphQL	27
2.6.1 - Esempio di query GraphQL vs REST	28
CAP. 3	29
3.1 - Analisi dei requisiti e casi d'uso	30
3.1.1 - Requisiti funzionali principali	30
3.1.2 - Requisiti non funzionali	30
3.1.3 - Casi d’uso principali	30
3.1.4 - Considerazioni finali	31
3.2 - Gestione del magazzino	31
3.2.1 - Flusso operativo lato front-end	32
3.2.2 - Struttura dati e logica lato back-end	33
3.2.3 - Automazioni e sicurezza del dato	33
3.2.4 - Integrazione con gli altri moduli	34
3.2.5 - Considerazioni finali	34
3.3 - Gestione del flusso di cassa	34
3.3.1 - Flusso operativo completo lato front-end – vendite, resi, sospesi e buoni	35

3.3.2 - Operazioni avanzate supportate	36
3.3.3 - Ruolo del back-end: logica e tracciamento	36
3.4 – Contabilità	37
3.4.1 - Visualizzazione e analisi dei conti	37
3.4.2 - Analisi di fatturato, acquisti e guadagni	38
3.4.3 - Inventario: valore e quantità delle scorte	39
3.4.4 - Back-end: generazione dei dati contabili	40
3.5 - Gestione dei media, dei metadati e delle informazioni per il web	41
3.6 - Pubblicazione e sincronizzazione con i canali di vendita web	42
3.6.1 - Pubblicazione su Shopify	42
3.6.2 - Pubblicazione su eBay	43
3.6.3 - Sincronizzazione con Amazon	44
3.6.4 - Task automatici di sincronizzazione e aggiornamento	45
CAP. 4.....	46
4.1 - Struttura del database.....	47
4.2 - Struttura delle api locali	48
4.3 - Struttura delle api web per lo store online (Shopify, eBay, Amazon).....	50
4.4 - Struttura delle api di stampa.....	51
4.5 - Task automatici in background di sincronizzazione con il web	52
4.6 - Hooks dalle piattaforme web	52
CAP. 5.....	54
5.1 - Organigramma del sito	55
5.1.1. - Homepage Dashboard	55
5.1.2 - Gestione Magazzino	55
5.1.3 - Gestione Assets.....	56
5.1.4 - Sistema Cassa.....	56
5.1.5 - Configurazione	56
5.2 - Impostazione e tema grafico.....	56
5.3 - Interfacciamento con le API del back-end	58
CAP. 6.....	59
6.1 - Versionamento del progetto tramite GIT	60
6.1.1 - Organizzazione delle branch	60
6.1.2 - Flusso di lavoro personale	60
6.1.3 - Vantaggi riscontrati	60

6.2 - Test automatici delle api del back	61
6.2.1 - Test unitari	61
6.2.2 - Test integrativi.....	61
6.2.3 - Test cross-browser (end-to-end)	61
6.2.4 - Test di performance	62
6.3 - Messa in produzione del sistema tramite Docker	62
6.4 - Creazione dello stack Docker per servire il sistema gestionale sulla rete interna	63
6.4.1 - Configurazione HTTPS con certificati gratuiti	63
6.4.2 - Servizi ausiliari per amministrazione	63
6.4.3 - Gestione volumi e persistenza	64
6.4.5 - Rete e sicurezza.....	64
6.5 - CI/CD per automatizzare i test e la messa in produzione	64
6.5.1 - Strumenti utilizzati	64
6.5.2 - Vantaggi per la gestione futura	65
CAP. 7.....	66
7.1 - Numeri e dimensioni del sistema	67
7.1.1 - Dati gestionali	67
7.1.2 - Struttura software.....	67
7.1.3 - Media e contenuti web	67
7.1.4 - Sincronizzazione e automazioni.....	68
7.1.5 - Utenti e sicurezza	68
7.2 - Performance del back-end e del front-end	68
7.2.1 - Performance del back-end	68
7.2.2 - Performance del front-end	69
7.2.3 - Performance complessiva e affidabilità.....	69

Elenco delle Figure

FIGURA 1: RIPARTIZIONE FATTURATO E-COMMERCE PER CANALI DI VENDITA	14
FIGURA 2: SERVER NODE.JS SENZA FRAMEWORK	20
FIGURA 3: SERVER NODE.JS CON EXPRESS.JS	20
FIGURA 4: JAVASCRIPT “VANILLA”	21
FIGURA 5: REACT COMPONENT	21
FIGURA 6: ACCESSO AI DATI SENZA ORM	23
FIGURA 7: UTILIZZO ORM DJANGO	23
FIGURA 8: ESEMPIO DJANGO	27
FIGURA 9: ESEMPIO RICHIESTA GRAPHQL	28
FIGURA 10: DIAGRAMMA A FLUSSO PER AGGIUNGERE ORDINI AL MAGAZZINO	31
FIGURA 11: PAGINA GESTIONE ORDINI	32
FIGURA 12: PAGINA DETTAGLIO DI UN ORDINE	33
FIGURA 13: DIAGRAMMA FLUSSO DI CASSA	35
FIGURA 14: ESEMPIO UTILIZZO PAGINA CASSA	35
FIGURA 15: DETTAGLI DI UN ACQUISTO SOSPESO	36
FIGURA 16: LISTA DEI BUONI	36
FIGURA 17: PAGINA DEI CONTI	38
FIGURA 18: ESEMPIO DEI DETTAGLI DI UN CONTO	38
FIGURA 19: PAGINA DEI GUADAGNI	39
FIGURA 20: PAGINA INVENTARIO	40
FIGURA 21: PAGINA GESTIONE IMMAGINI PRODOTTO	41
FIGURA 22: DETTAGLI SHOPIFY DI UN PRODOTTO	43
FIGURA 23: DETTAGLI PRODOTTO EBAY	44
FIGURA 24: OPZIONI DI TASK AUTOMATICI DA AVVIARE	45
FIGURA 25: UML DATABASE	47
FIGURA 26: DIAGRAMMA API WAREHOUSE	49
FIGURA 27: DIAGRAMMA API WEB	51
FIGURA 28: DIAGRAMMA API STAMPA	52
FIGURA 29: DIAGRAMMA DI SEQUENZA PER I WEBHOOKS	53
FIGURA 30: ORGANIGRAMMA DEL SITO	55
FIGURA 31: PAGINA DETTAGLIO DI UN PRODOTTO	57
FIGURA 32: HOMEPAGE	57
FIGURA 33: CONFIGURAZIONE DOCKER	63

Elenco delle Tabelle

TABELLA 1: ANALISI DEI REQUISITI	30
--	----

1

Cap. 1

Introduzione

Questa tesi presenta lo sviluppo di un sistema gestionale realizzato per una piccola impresa specializzata nella vendita di abbigliamento e articoli militari. Questo progetto nasce dall'esigenza di digitalizzare le operazioni quotidiane di un negozio di abbigliamento attraverso un sistema gestionale unificato. Il sistema per sincronizzare ordini e acquisti provenienti da diversi canali online Shopify, Amazon ed eBay, gestire il magazzino e la cassa, stampare scontrini ed etichette, e monitorare l'andamento delle vendite. L'obiettivo è supportare la transizione digitale della PMI, migliorando efficienza, precisione e controllo.

Durante la fase iniziale, ho osservato che l'azienda affrontava problemi ricorrenti: errori di sincronizzazione tra i canali, errori nelle quantità dovuti a una gestione manuale delle giacenze e difficoltà nell'analisi dei dati di vendita. Questi problemi limitavano la capacità dell'azienda e mettevano in difficoltà il personale.

Da questa esigenza è nata l'idea di sviluppare un sistema gestionale integrato, capace di unificare la logistica, sincronizzare in tempo reale gli ordini nel negozio su Shopify, Amazon ed eBay e fornire un'unica interfaccia per monitorare scorte, evadere ordini e analizzare performance. Il mio sistema, basato su Django per il back-end e React per il front-end, è stato progettato proprio per ovviare ai problemi elencati in precedenza e anche ad altri.

1.1 - Globalizzazione e necessità per le piccole realtà alla vendita online

La globalizzazione ha radicalmente trasformato il contesto competitivo delle imprese, introducendo nuove sfide ma anche enormi opportunità. Grazie alla riduzione delle barriere geografiche, economiche e tecnologiche, oggi anche una piccola impresa può accedere a mercati internazionali senza necessariamente sostenere i costi di una presenza fisica all'estero. Il commercio elettronico rappresenta uno degli strumenti più potenti in questo scenario: consente di vendere prodotti a clienti in ogni parte del mondo, 24 ore su 24, superando i limiti spazio-temporali tipici del commercio tradizionale.

Nel 2023, il 78% degli utenti Internet in Italia ha effettuato almeno un acquisto online, secondo i dati dell'ISTAT, in crescita rispetto al 74% del 2022. Questo trend è ancora più marcato tra le generazioni più giovani, abituate a esperienze di acquisto fluide, rapide e personalizzate. I consumatori moderni si aspettano non solo la possibilità di acquistare da qualsiasi luogo e in qualsiasi momento, ma anche un'esperienza utente coerente tra diversi canali (web, app mobile, social media). In questo contesto, non offrire un canale di vendita online equivale a rinunciare a una quota significativa di mercato, che verrà inevitabilmente intercettata da competitor più tecnologicamente preparati, spesso anche esteri.

La digitalizzazione delle PMI italiane è ancora limitata, soprattutto se confrontata con la media europea. Secondo i dati Eurostat 2023, solo il 18,5% delle piccole imprese italiane dispone di un canale e-commerce attivo, contro una media dell'UE del 22,2%. Questo gap evidenzia una vulnerabilità sistemica: le imprese non digitalizzate rischiano di essere escluse dalle catene globali del valore, dove rapidità, tracciabilità e automazione dei processi sono ormai prerequisiti. Ad esempio, grandi operatori della logistica e della distribuzione richiedono sistemi integrati per l'evasione degli ordini, la gestione delle giacenze e la tracciabilità dei pacchi, pena l'impossibilità di collaborare in modo efficiente.

La crescente frammentazione dei canali di vendita complica ulteriormente la situazione. Oltre al proprio sito e-commerce, molte PMI devono gestire piattaforme marketplace (Amazon, eBay, Etsy), social commerce (Instagram, TikTok, Facebook Shops), e persino canali di messaggistica (WhatsApp Business, Telegram). Ogni piattaforma presenta requisiti tecnici e strategici differenti:

- Amazon, ad esempio, impone SLA stringenti sulla spedizione, penalizza i ritardi e richiede un inventario aggiornato in tempo reale.
- Shopify, al contrario, offre maggiore autonomia nella gestione del negozio online, ma richiede conoscenze specifiche di marketing digitale e investimenti pubblicitari per generare traffico.
- Instagram e altri social network garantiscono visibilità immediata, ma il loro funzionamento è strettamente legato ad algoritmi mutevoli e strategie di contenuto dinamiche.

In assenza di strumenti integrati, la gestione di questi canali avviene spesso in modo manuale, attraverso fogli Excel, e-mail e interventi umani dispendiosi in termini di tempo e soggetti a errori. Ciò genera problematiche ricorrenti.

In un contesto sempre più orientato al time-to-market e alla customer satisfaction, tali inefficienze possono compromettere seriamente la reputazione dell'azienda e ridurre la fidelizzazione dei clienti. Una risposta concreta a questa complessità è rappresentata da sistemi informatici centralizzati in grado di gestire e sincronizzare le operazioni tra i diversi canali. Il sistema gestionale sviluppato in questo progetto nasce proprio con questo obiettivo: fornire un unico punto di controllo per tutte le attività legate alla vendita online. Tra i principali vantaggi:

- unificazione del magazzino: tutte le giacenze vengono monitorate in tempo reale da un'unica fonte di verità
- sincronizzazione automatica degli ordini: ogni vendita comporta un aggiornamento in tempo reale delle disponibilità su tutti i canali collegati
- analisi in tempo reale delle performance di vendita, con cruscotti personalizzabili e filtri per canale, categoria e periodo
- L'automazione consente di intervenire rapidamente su eventuali anomalie, limitando il margine di errore

Questa soluzione consente all'azienda di adattarsi gradualmente alla crescita senza dover ripensare da zero l'infrastruttura. Questo è particolarmente rilevante per le imprese italiane che intendono espandersi nei mercati esteri.

Per molte piccole realtà, digitalizzazione e globalizzazione sono ormai condizioni indispensabili per sopravvivere nel mercato. L'e-commerce rappresenta la porta d'accesso ai mercati globali, ma richiede una gestione strutturata, integrata e tecnologicamente evoluta. Solo attraverso l'adozione di strumenti informatici adeguati, come quelli descritti in questo progetto, le PMI possono realmente competere ad armi pari con realtà più grandi e preparate, mantenendo al tempo stesso la flessibilità e la specializzazione che le caratterizzano.

1.2 - Canali di vendita online, frazionamento del mercato

In Italia, il 40% degli acquisti online avviene su siti proprietari. Questo significa che un'azienda che vende su Shopify o su un sito WordPress ha piena libertà di definire l'esperienza utente, i margini di profitto e la comunicazione del brand. Ma c'è un rovescio della medaglia: mantenere un e-commerce richiede investimenti continui in marketing, SEO e manutenzione tecnica. Un esempio pratico: per far crescere il traffico organico, una PMI deve spendere tempo a ottimizzare i contenuti o affidarsi a agenzie specializzate, mentre su Amazon basterebbe pagare per apparire tra i primi risultati.

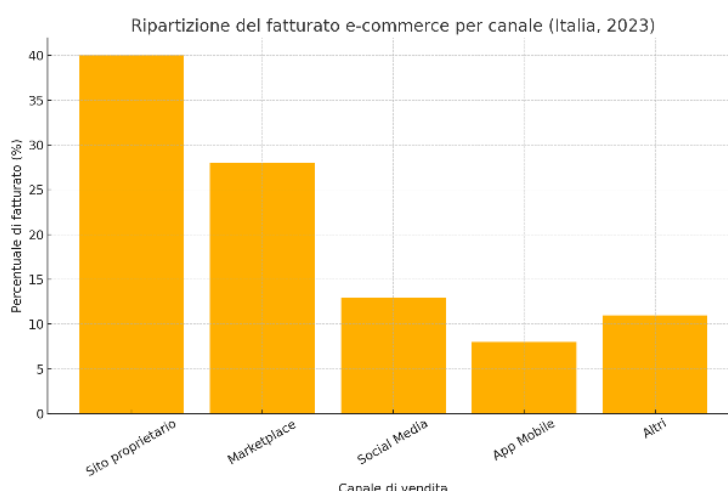


Figura 1: ripartizione fatturato e-commerce per canali di vendita

I marketplace, che pesano per il 28% del mercato, offrono una scorciatoia evidente: milioni di utenti già attivi, checkout integrato, gestione dei resi semplificata. Tuttavia, chi vende su eBay o Amazon paga commissioni che possono arrivare al 15-20% del prezzo finale, senza contare i vincoli contrattuali (es. politiche di pricing che vietano di essere più economici altrove).

Un altro esempio è il social commerce, con il 13% del totale, è il fronte più dinamico. Instagram e TikTok non sono più solo strumenti di visibilità: permettono acquisti diretti dal feed, con tool integrati come i "Checkout Button". Il vantaggio è enorme per chi sa sfruttare il potere dei creator. Ma c'è un rischio: la dipendenza dagli algoritmi. Se un giorno TikTok cambia le regole dell'engagement, un'azienda che ha investito tutto in quel canale si ritrova a zero.

Infine esistono le app mobile che costituiscono l'8% del mercato italiano, dedicato alle vendite tramite app native, uno strumento spesso sottovalutato. Un cliente che ha la tua app installata riceve notifiche push, accede a promozioni esclusive e accumula punti fedeltà: è un rapporto più diretto rispetto al marketplace. Tuttavia, sviluppare e aggiornare un'app richiede competenze tecniche che molte PMI non hanno in-house.

Alla fine, la scelta dei canali diventa un equilibrio precario. Investire troppo su un unico canale espone a rischi; diversificare comporta costi di gestione. Un sistema gestionale integrato permette a una PMI di testare nuovi canali senza compromettere il controllo sugli ordini, sugli stock e sui dati. Senza di essa, ogni decisione strategica diventa un passo nel buio.

1.3 - Gestione di magazzini dislocati e canali di vendita eterogenei tramite un unico gestionale

Quando un'azienda opera su più fronti, con magazzini fisici distribuiti in diverse città e vendite su Shopify, Amazon, eBay e social commerce, mantenere una gestione ordinata e sincronizzata delle operazioni diventa una sfida quotidiana. Senza un sistema unificato, anche un piccolo errore può generare conseguenze a catena: un prodotto potrebbe risultare esaurito su un canale ma ancora acquistabile altrove, oppure si rischia di dover aggiornare manualmente decine di cataloghi e liste di stock, con un alto margine di errore e dispendio di tempo.

Per affrontare questi problemi in modo strutturato, sempre più PMI stanno investendo in soluzioni come i *Warehouse Management System (WMS)*, ovvero software per la gestione intelligente dei magazzini. In Italia, il mercato dei WMS è in rapida crescita, con un tasso annuo medio del 13,7%. Non si tratta semplicemente di sostituire un foglio Excel, ma di implementare un sistema capace di automatizzare il monitoraggio delle scorte, minimizzare il lavoro manuale e aumentare l'affidabilità delle operazioni. Tuttavia, un WMS da solo non è sufficiente per gestire la complessità del commercio multicanale moderno. È qui che entra in gioco il concetto di *Multichannel Order Management (MOM)*, ovvero piattaforme che centralizzano e armonizzano la ricezione e gestione degli ordini provenienti da fonti diverse. Ogni marketplace ha regole proprie: Amazon impone formati e tempistiche precise, Shopify consente maggiore personalizzazione, eBay ha dinamiche autonome. Gestire tutto manualmente è insostenibile, per questo un sistema MOM permette di automatizzare il flusso di lavoro tra i canali, offrendo un'unica

interfaccia da cui monitorare, processare e tracciare ogni ordine, riducendo drasticamente il rischio di disguidi o duplicazioni.

Oltre alla logistica, esistono anche leve strategiche per aumentare la visibilità online e incentivare le vendite: tra queste, l'uso mirato di sconti digitali e promozioni dedicate agli acquisti via e-commerce. Offrire sconti esclusivi su piattaforme come Shopify o Amazon permette non solo di attrarre nuovi clienti, ma anche di fidelizzare quelli esistenti e spingerli verso il canale online, dove la gestione è automatizzata e più efficiente. In questo modo, si ottiene un duplice vantaggio: da un lato si alleggerisce il carico operativo del punto vendita fisico, dall'altro si amplifica la portata del negozio, raggiungendo una nuova clientela che altrimenti non verrebbe a conoscenza del punto vendita.

Un ulteriore valore aggiunto risiede nella capacità di analizzare i dati in modo avanzato. Un pannello di controllo unificato non si limita a mostrare la disponibilità a magazzino, ma permette di prevedere con precisione quando riordinare determinati articoli, grazie all'analisi di trend stagionali, preferenze dei clienti e picchi di vendita. In un contesto in cui gli investimenti nel settore logistico sono diminuiti del 42%, l'ottimizzazione dei costi operativi e delle scorte diventa cruciale per la sopravvivenza e la competitività delle piccole imprese.

Per questo motivo, le PMI che vogliono espandersi oltre i confini locali e affrontare il mercato globale, devono adottare un sistema integrato per la gestione delle vendite e della logistica. La combinazione di automazione, analisi dati e promozioni mirate è oggi la chiave per crescere in un ambiente digitale sempre più competitivo.

1.4 – Perché sviluppare un sistema gestionale personalizzato e non usare soluzioni già esistenti

Una delle prime valutazioni affrontate nel progetto è stata se utilizzare un software gestionale già disponibile sul mercato – come SAP, Oracle NetSuite, Odoo o soluzioni SaaS verticali per e-commerce – oppure sviluppare un sistema gestionale su misura. Sebbene le soluzioni esistenti offrano numerose funzionalità e siano spesso percepite come “chiavi in mano”, presentano anche diversi limiti quando applicate a una piccola impresa con esigenze specifiche.

In primo luogo, le PMI faticano ad adattarsi a software pensati per grandi aziende. I software enterprise come SAP o NetSuite sono potenti ma anche complesse, costose e spesso sovradimensionate. Richiedono formazione, consulenza tecnica e costi di licenza o abbonamento difficilmente sostenibili per un'attività con budget e personale limitati. In alcuni casi, è proprio l'eccesso di funzioni inutilizzate a complicare l'adozione e rallentare il lavoro quotidiano.

In secondo luogo, molte soluzioni SaaS sono pensate per processi standardizzati. Tuttavia, ogni PMI ha proprie dinamiche: nel caso del negozio oggetto di questo progetto, vi erano richieste molto precise legate alla stampa degli scontrini e delle etichette, all'integrazione simultanea con Shopify, Amazon ed eBay, alla gestione di sconti su misura e all'unificazione della contabilità e della cassa fisica. Personalizzare un gestionale preesistente per adattarlo a questi flussi avrebbe comportato comunque interventi costosi e spesso limitati dalle API fornite dal vendor.

Un altro aspetto critico è la gestione dei dati. I software SaaS archiviano i dati su server esterni e, pur offrendo garanzie di sicurezza, limitano l'accesso diretto alle informazioni grezze e alla logica di elaborazione. In un sistema personalizzato, invece, l'azienda ha pieno controllo su database, report, backup e flussi interni, riducendo il rischio di lock-in tecnologico e aumentando la trasparenza.

Lo sviluppo su misura ha permesso di costruire un gestionale flessibile, in linea con le necessità specifiche dell'azienda e adattabile a future evoluzioni operative. La possibilità di testare, iterare e integrare nuove funzionalità (come la gestione degli stock distribuiti o la contabilità avanzata) è risultata fondamentale per garantire la sostenibilità a lungo termine del progetto.

Questa decisione si è rivelata strategica per l'azienda, andando oltre gli aspetti puramente tecnici. Il risultato è un sistema integrato con i principali canali digitali, progettato per snellire concretamente le attività operative.

2

Cap. 2

Linguaggi di programmazione
back-end e front-end

Dopo aver definito il problema della PMI con cui ho collaborato (gestione manuale di magazzino e sincronizzazione tra canali), ecco come ho affrontato la parte tecnica. Il progetto si basa su due pilastri: Django per il back-end e React per il front-end , scelti per la loro capacità di semplificare lo sviluppo di sistemi complessi come questo.

Per gestire il database, ho utilizzato un ORM. Per la comunicazione tra back-end e front-end, ho optato per RESTful API e per la comunicazione con i canali di vendita online ho utilizzato le GraphQL.

Ogni decisione è stata guidata da un obiettivo pratico: rendere il sistema gestionale scalabile, manutenibile e accessibile anche a chi non ha competenze tecniche avanzate. In questo capitolo viene analizzato il perché ho scelto queste tecnologie.

2.1 - Vantaggi nell'utilizzare framework sia per back-end che il front-end

Un framework è una struttura software che fornisce una base riutilizzabile e organizzata per lo sviluppo di applicazioni. A differenza di una libreria, che offre funzioni da richiamare secondo necessità, un framework definisce un flusso di lavoro prestabilito: è il framework a chiamare il codice dello sviluppatore, secondo un'architettura precisa. Questo approccio, noto come *inversion of control*, consente di ridurre la complessità e uniformare il modo in cui vengono sviluppate le applicazioni.

Nel contesto dello sviluppo web moderno, l'utilizzo di framework rappresenta una pratica consolidata sia lato back-end che front-end. Essi consentono di velocizzare lo sviluppo e migliorare la manutenzione del codice, inoltre permettono di ridurre il rischio di errori e soprattutto di adottare facilmente pattern architetturali collaudati (come MVC, MVVM o component-based).

I framework back-end forniscono strumenti integrati per gestire routing, accesso ai dati, autenticazione, sicurezza, validazione delle richieste, gestione degli errori e molto altro. Questo riduce drasticamente la quantità di codice da scrivere manualmente. Framework noti come **Express.js**, **Spring**, **Django**, **Laravel** e **Ruby on Rails** sono largamente adottati per la loro capacità di fornire un'infrastruttura solida e modulare.

Secondo l'indagine Stack Overflow Developer Survey 2024:

- oltre il 50% degli sviluppatori back-end utilizza Node.js, spesso insieme a framework come Express o NestJS;
- Spring Boot è ancora dominante in ambito enterprise;
- i framework Python come Django e FastAPI hanno visto una crescita costante grazie alla loro semplicità e integrazione con strumenti per il machine learning e i microservizi.

Sul lato front-end, i framework e le librerie moderne hanno rivoluzionato lo sviluppo delle interfacce utente, introducendo concetti come il rendering reattivo, la gestione centralizzata dello stato, e l'aggiornamento efficiente del DOM. Strumenti come **React**, **Vue.js**, **Angular** e **Svelte** permettono di sviluppare interfacce modulari, dinamiche e facilmente scalabili.

Sempre secondo la Stack Overflow Survey 2024:

- **React** è la libreria front-end più usata, con il **47%** di adozione tra gli sviluppatori;
- **Vue.js** è apprezzato per la sua leggerezza e semplicità, soprattutto nei progetti di piccola-media scala;
- **Angular**, con la sua architettura completa, continua ad essere preferito in contesti enterprise strutturati.

L'ecosistema che accompagna questi framework è altrettanto rilevante: **npm** (per JavaScript) e **pip** (per Python) contano milioni di pacchetti che permettono di estendere rapidamente le funzionalità, favorendo il riuso del codice e l'integrazione con altri sistemi.

Utilizzare un framework nello sviluppo di un'applicazione web, sia sul front-end che sul back-end, non è solo una scelta tecnica, ma strategica. Significa affidarsi a strumenti collaudati, mantenuti attivamente dalle community, e capaci di evolversi insieme alle tecnologie e alle esigenze del mercato. Per un

progetto moderno, questa scelta rappresenta una garanzia di efficienza, scalabilità e sostenibilità nel lungo termine.

2.1.1 - Esempio “Prima e Dopo” l’adozione di un framework Back-end

Criticità prima di utilizzare un framework su node.js:

- Gestione manuale del parsing del body e degli header.
- Nessuna validazione automatica dei dati.
- Routing hard-coded, difficile da estendere.

```
// server.js
const http = require('http');
const todos = [];
const server = http.createServer((req, res) => {
  if (req.method === 'POST' && req.url === '/api/todos') {
    let body = '';
    req.on('data', chunk => body += chunk);
    req.on('end', () => {
      const item = JSON.parse(body).text;
      todos.push({ id: todos.length + 1, text: item });
      res.writeHead(201, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify(todos));
    });
  }
  else if (req.method === 'GET' && req.url === '/api/todos') {
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify(todos));
  }
  else {
    res.writeHead(404);
    res.end();
  }
});
server.listen(3000, () => console.log('Server in ascolto su 3000'));
```

Figura 2: server Node.js senza framework

Benefici nell’implementare un framework come Express.js:

- Parsing JSON e validazione base out-of-the-box.
- Routing dichiarativo, facilmente estendibile.
- Middleware per logging, sicurezza, CORS ecc.

```
// app.js
const express = require('express');
const app = express();
const todos = [];
app.use(express.json()); // parsing automatico

app.post('/api/todos', (req, res) => {
  const { text } = req.body;
  if (!text) return res.status(400).json({ error: 'Testo obbligatorio' });
  todos.push({ id: todos.length + 1, text });
  res.status(201).json(todos);
});

app.get('/api/todos', (req, res) => {
  res.json(todos);
});
app.listen(3000, () => console.log('Server Express su 3000'));
```

Figura 3: server Node.js con Express.js

2.1.2 - Esempio “Prima e Dopo” l’adozione di un framework Front-end

Criticità nella prima versione senza l’utilizzo di un framework:

- Manipolazione diretta del DOM, codice verboso.
- Difficile gestire aggiornamenti incrementali.

```
<!-- index.html -->
<ul id="todo-list"></ul>
<script>
  const todos = ['Comprare latte', 'Studiare per l'esame'];
  const listEl = document.getElementById('todo-list');

  // render manuale
  todos.forEach(item => {
    const li = document.createElement('li');
    li.textContent = item;
    listEl.appendChild(li);
  });
</script>
```

Figura 4: JavaScript “vanilla”

Benefici nell’utilizzare un React Component:

- Aggiornamenti efficienti tramite virtual DOM.
- Componenti riutilizzabili e testabili.
- Chiarezza sintattica e separazione delle responsabilità.

```
// TodoList.jsx
import React from 'react';

function TodoList({ todos }) {
  return (
    <ul>
      {todos.map((item, idx) => (
        <li key={idx}>{item}</li>
      ))}
    </ul>
  );
}

export default TodoList;
```

Figura 5: React Component

2.2 - Approccio moderno per la strutturazione database tramite ORM

Nel contesto dello sviluppo software moderno, l’Object-Relational Mapping (ORM) rappresenta un approccio sempre più diffuso per la gestione della persistenza dei dati. Un ORM è una tecnica che consente di interfacciarsi con un database relazionale attraverso un linguaggio orientato agli oggetti, astrattizzando il livello SQL e permettendo agli sviluppatori di interagire con i dati come se fossero normali oggetti del linguaggio di programmazione utilizzato.

A livello concettuale, l’ORM si occupa di mappare le tabelle del database in classi, e ogni riga (record) in un oggetto. Le colonne corrispondono agli attributi dell’oggetto, e le relazioni tra tabelle (foreign key, join) vengono modellate tramite attributi speciali (associazioni uno-a-uno, uno-a-molti, molti-a-molti). In questo modo, si può scrivere codice per leggere, aggiornare, creare o cancellare dati (CRUD) senza dover scrivere direttamente query SQL.

Uno dei principali vantaggi è l'incremento della produttività. Scrivere codice orientato agli oggetti è generalmente più rapido, leggibile e manutenibile rispetto alla scrittura manuale di query SQL, soprattutto in progetti di medie o grandi dimensioni. L'ORM permette inoltre di astrarre il database sottostante: lo stesso codice può funzionare sia su PostgreSQL, che su MySQL, SQLite o altri motori, con minime modifiche, semplificando il processo di portabilità.

Un altro aspetto centrale è la sicurezza. L'uso corretto di un ORM riduce drasticamente la possibilità di vulnerabilità legate a SQL injection, una delle tecniche di attacco più comuni nei software legacy. Poiché i valori vengono passati come parametri sicuri anziché concatenati nelle stringhe SQL, il rischio di esecuzione di codice maligno all'interno delle query è fortemente mitigato.

Dal punto di vista architetturale, l'ORM contribuisce a una maggiore coerenza del modello di dominio: il codice riflette direttamente la logica di business, con entità chiaramente definite e relazioni esplicite. Questo approccio facilita l'integrazione con pattern come MVC (Model-View-Controller), Repository pattern o Domain Driven Design (DDD), dove i modelli del database si integrano naturalmente nella struttura applicativa.

Un ulteriore vantaggio è la gestione delle migrazioni: gli ORM moderni forniscono strumenti per versionare lo schema del database e applicare modifiche in modo controllato e incrementale (migration system), riducendo il rischio di errori in fase di aggiornamento.

Nonostante i numerosi vantaggi, l'ORM presenta anche alcuni limiti. Uno dei più citati è la perdita di controllo fine sulle prestazioni. Le query generate automaticamente, se non ottimizzate o se eseguite in contesti complessi (join multipli, filtri avanzati, sottoquery), possono essere meno efficienti rispetto a quelle scritte manualmente da un DBA esperto. Questo problema è particolarmente visibile quando si gestiscono grandi quantità di dati o operazioni ad alta frequenza, dove la latenza di una query può avere un impatto diretto sull'usabilità del sistema.

In alcuni casi, l'astrazione dell'ORM può nascondere la complessità reale del database, inducendo gli sviluppatori a ignorare aspetti importanti come gli indici, i piani di esecuzione delle query o il locking dei record. In situazioni in cui è necessaria un'ottimizzazione molto spinta, oppure si lavora con caratteristiche specifiche di un motore, potrebbe essere preferibile un approccio ibrido o manuale.

Alcuni ORM impongono anche una certa curva di apprendimento, soprattutto in presenza di relazioni complesse o funzionalità avanzate. Gli sviluppatori devono comprendere bene come il sistema costruisce le query, gestisce le transazioni e interagisce con la cache dei risultati per evitare comportamenti inattesi o inefficienze nascoste.

Gli ORM sono oggi ampiamente usati nello sviluppo applicativo per velocizzare il lavoro e rendere più chiaro il codice, soprattutto in contesti agili o a team ridotto, dove la velocità di sviluppo e la chiarezza del codice sono prioritarie. Pur non essendo la soluzione ideale in tutti i casi, il paradigma ORM consente di scrivere software più sicuro, coerente e facile da mantenere, contribuendo alla stabilità e alla scalabilità del sistema nel lungo termine.

Nei prossimi sottocapitoli verrà illustrato più nel dettaglio l'uso concreto dell'ORM fornito da Django, con esempi applicativi legati alla gestione delle entità chiave del progetto.

2.2.1 - Esempio “Prima e Dopo” l’adozione di un ORM

Le criticità che si possono presentare prima di utilizzare un ORM possono essere: la concatenazione di stringhe, la mancanza di migrazioni automatiche, e il codice non riutilizzabile.

```
import sqlite3

conn = sqlite3.connect('db.sqlite3')
cur = conn.cursor()

# Inserimento
data = ('Mario Rossi', 'mario@example.com')
cur.execute(f"INSERT INTO users (name, email) VALUES ('{data[0]}', '{data[1]}')")
conn.commit()

# Lettura
cur.execute("SELECT id, name, email FROM users WHERE active = 1")
rows = cur.fetchall()
for r in rows:
    print(r)
```

Figura 6: accesso ai dati senza ORM

I benefici che si possono avere quando si implementa un ORM sono: la protezione automatica da injection, l’interfaccia a oggetti, le migrazioni gestite da manage.py makemigrations / migrate, e i metodi riutilizzabili.

```
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(unique=True)
    active = models.BooleanField(default=True)

# script di utilizzo
from myapp.models import User

# Inserimento
u = User.objects.create(name='Mario Rossi', email='mario@example.com')

# Lettura
active_users = User.objects.filter(active=True)
for user in active_users:
    print(user.id, user.name, user.email)
```

Figura 7: utilizzo ORM django

2.3 - Django framework

Django è un framework web open-source scritto in Python, progettato per facilitare lo sviluppo rapido e sicuro di applicazioni web complesse. È ampiamente utilizzato per la sua capacità di gestire molti aspetti critici di un'applicazione, come la gestione dei dati, la sicurezza e la scalabilità.

Uno dei punti di forza di Django è la sua struttura chiara e organizzata. Il framework adotta un approccio basato su model-template-view (MTV), dove i modelli definiscono la struttura dei dati nel database, i template gestiscono la presentazione dei dati agli utenti e le view controllano il flusso di dati e la logica dell'applicazione.

Ad esempio, nell'applicazione che ho progettato per la PMI, Django permette di definire i modelli del magazzino e degli ordini nel file models.py. Questi modelli rappresentano i dati nel database, come i dettagli degli ordini e le scorte disponibili. Django fornisce un'interfaccia admin nativa che consente alla PMI di gestire questi dati senza la necessità di scrivere codice aggiuntivo per l'admin panel.

Per le API, Django REST Framework (DRF) è estremamente utile. Con poche righe di codice, è possibile definire un Serializer per specificare come i dati devono essere serializzati e deserializzati tra il database

e i formati di trasmissione come JSON. Ad esempio, quando un ordine arriva da eBay, DRF gestisce automaticamente la validazione dei dati per garantire che siano coerenti prima di salvarli nel database, evitando errori come l'ordine di quantità non disponibili.

Dal punto di vista della sicurezza, Django protegge l'applicazione da vulnerabilità comuni come SQL injection e CSRF (Cross-Site Request Forgery) attraverso meccanismi incorporati. Questo è fondamentale per una PMI che vuole concentrarsi sulla gestione logistica senza doversi preoccupare di minacce informatiche.

Infine, la struttura modulare di Django facilita l'aggiunta di nuove funzionalità. Il sistema è suddiviso in moduli funzionali, ciascuno dedicato a un ambito specifico come ordini o inventario. Questo permette di aggiungere nuove funzioni senza riscrivere il codice esistente, mantenendo il sistema scalabile e manutenibile nel tempo.

2.4 - React Framework

React è una libreria JavaScript open-source sviluppata da Meta per la creazione di interfacce utente (UI), in particolare interfacce dinamiche e interattive per applicazioni web single-page (SPA). A differenza di framework completi come Angular o Vue, React si concentra esclusivamente sulla vista, lasciando la gestione della logica e del routing ad altre librerie opzionali. È basato sul concetto di componenti riutilizzabili e sul virtual DOM, un sistema che ottimizza gli aggiornamenti dell'interfaccia utente minimizzando l'interazione diretta con il DOM reale del browser.

Una delle sue caratteristiche principali è la programmazione dichiarativa: invece di descrivere passo dopo passo come aggiornare l'interfaccia, si dichiara come dovrebbe apparire in base allo stato corrente dell'applicazione. Quando questo stato cambia, React si occupa automaticamente di aggiornare solo le parti necessarie dell'interfaccia, in modo efficiente.

Per il front-end del sistema gestionale, ho scelto React per una ragione pratica: permette di costruire interfacce dinamiche. La PMI con cui ho collaborato aveva bisogno di un sistema che mostrasse in tempo reale lo stato delle scorte, gli ordini in arrivo e i dati di vendita. Con React, ogni volta che un prodotto veniva aggiornato nel database, l'interfaccia si aggiornava dinamicamente grazie al virtual DOM, senza dover ricaricare la pagina. Questo ha ridotto il rischio di errori (es. visualizzazione di stock non sincronizzato) e migliorato l'efficienza operativa.

Un altro punto a favore è stato il paradigma a componenti. Invece di scrivere codice per ogni parte dell'interfaccia, ho creato componenti riutilizzabili. Questo ha reso il codice più leggibile e scalabile. Ogni componente può contenere stato locale (tramite `useState`) e logica di ciclo di vita (tramite `useEffect`), rendendo facile incapsulare comportamenti complessi.

Per gestire degli stati globali come i dati dell'utente loggato o giacenze aggiornate, ho usato React Context. Il vantaggio è stato che ogni volta un ordine veniva sincronizzato dal back-end, il front-end rifletteva il cambiamento in tempo reale. Per la PMI, questo significava monitorare le scorte senza dover controllare manualmente diversi canali, risparmiando ore di lavoro.

Infine, React ha reso semplice l'integrazione con le API del back-end. Grazie a librerie come Axios, ho potuto chiamare gli endpoint Django per recuperare dati su stock o inviare aggiornamenti. E con React Router, navigare tra le varie pagine, senza dover gestire manualmente gli URL. In sintesi, React non è solo un framework per creare interfacce: è stato lo strumento che ha reso il front-end reattivo, manutenibile e accessibile. Per la PMI, questo ha significato un'esperienza utente coerente; per me, sviluppatore, un codice più efficiente da gestire e poco propenso a bug.

2.5 - Restful API

Le RESTful API (Representational State Transfer) rappresentano uno degli standard più diffusi per la comunicazione tra client e server in applicazioni web moderne. Introdotto da Roy Fielding nel 2000, il paradigma REST si basa su un insieme di principi architetturali che sfruttano le convenzioni del protocollo HTTP per esporre risorse in modo semplice, leggibile e scalabile.

Una REST API struttura l'interfaccia come una serie di endpoint associati a risorse (ad esempio: /users, /products, /orders), e utilizza HTTP per eseguire operazioni CRUD:

- GET per recuperare dati,
- POST per creare nuove risorse,
- PUT o PATCH per aggiornarle,
- DELETE per eliminarle.

Questi endpoint sono progettati per essere *resource-oriented*, ovvero ogni URL rappresenta una risorsa univoca, identificabile e manipolabile attraverso operazioni ben definite. Questa organizzazione rende l'API intuitiva da usare anche per sviluppatori non coinvolti direttamente nello sviluppo del back-end.

Ogni richiesta REST è *stateless*: ciò significa che il server non mantiene memoria delle interazioni precedenti. Ogni chiamata contiene tutte le informazioni necessarie (header di autenticazione, parametri, corpo della richiesta) per essere processata in modo indipendente. Questa caratteristica rende REST particolarmente adatto ad architetture scalabili e distribuite, in cui le richieste possono essere bilanciate su più istanze di server senza problemi di consistenza.

Il formato più comune per la rappresentazione delle risposte REST è il JSON (JavaScript Object Notation), apprezzato per la sua sintassi leggera, la facile leggibilità umana e la compatibilità nativa con JavaScript e la maggior parte dei linguaggi di programmazione moderni. Tuttavia, REST è agnostico rispetto al formato dei dati: in contesti specifici, può anche restituire XML, YAML o formati binari personalizzati.

REST promuove inoltre l'uso di codici di stato HTTP (200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Internal Server Error, ecc.) per comunicare l'esito delle operazioni in maniera coerente e standardizzata. Questo contribuisce a migliorare il debugging, la gestione degli errori e la manutenzione del codice client.

Secondo il report *State of API 2023* di Postman, oltre l'89% delle organizzazioni che sviluppano API continua a utilizzare REST come approccio principale, anche laddove vengono introdotti nuovi paradigmi come GraphQL o gRPC. Inoltre, il 68% degli sviluppatori back-end ha indicato REST come la tecnologia API più conosciuta e implementata nei progetti professionali, rispetto al 24% di GraphQL e al 7% di gRPC. Questo conferma il ruolo centrale di REST nel panorama tecnologico attuale.

I principali vantaggi delle REST API includono:

- **Compatibilità con tool standard:** strumenti come Postman, Swagger/OpenAPI, Insomnia o Paw permettono di testare, documentare e validare facilmente le chiamate REST.
- **Supporto nativo da parte dei browser:** grazie alla compatibilità diretta con XMLHttpRequest o fetch(), REST è facilmente utilizzabile nei front-end basati su JavaScript.
- **Caching semplificato:** REST può sfruttare a pieno le funzionalità di caching HTTP (tramite header come Cache-Control, ETag e Last-Modified), migliorando le performance in lettura.
- **Flessibilità architetturale:** la separazione client-server e l'adozione del protocollo HTTP consentono un deployment agile in ambienti distribuiti, ad esempio in architetture a microservizi.
- **Estensibilità e versionamento:** REST può facilmente gestire versioni diverse della stessa API (es. /api/v1/orders) per garantire retrocompatibilità con sistemi legacy.

Un ulteriore vantaggio significativo è la documentabilità. L'utilizzo del formato OpenAPI (in precedenza noto come Swagger) consente di definire la struttura dell'API in modo formale e leggibile. Questo facilita l'integrazione tra team diversi (es. front-end, back-end) e permette la generazione automatica di:

- client SDK per varie piattaforme (Python, Java, JavaScript, ecc.),
- interfacce grafiche esplorabili,
- test automatici di regressione.

Nel contesto del mio progetto, le REST API sono state fondamentali per la comunicazione tra il front-end React e il back-end Django. L'uso del framework Django REST Framework (DRF) ha semplificato la creazione di endpoint robusti, con validazioni automatiche, gestione coerente degli errori e autenticazione integrata.

REST ha inoltre consentito di integrare servizi esterni come eBay. Questo marketplace offre proprie REST API che, seppur con variazioni nei parametri e nell'autenticazione, si basano sugli stessi principi: risorse accessibili tramite URL, verbi HTTP standard, risposte in JSON e comportamento stateless.

In sintesi, le RESTful API hanno offerto una base solida, standardizzata e ben supportata per lo sviluppo dell'infrastruttura software. La loro adozione ha favorito la modularità del progetto, semplificato lo sviluppo front-back, reso il sistema facilmente integrabile con servizi esterni e garantito una documentazione chiara per il futuro mantenimento.

2.5.1 - Esempio in Django Rest Framework

Con queste poche righe, Django Rest Framework espone automaticamente tutti gli endpoint REST principali per la risorsa Product, incluso il supporto per autenticazione, validazione e serializzazione in formato JSON.

- **GET /api/products** → restituisce la lista dei prodotti
- **GET /api/products/12** → restituisce i dati del prodotto con ID 12
- **POST /api/products** → crea un nuovo prodotto
- **PUT /api/products/12** → aggiorna completamente il prodotto 12
- **PATCH /api/products/12** → aggiorna parzialmente il prodotto
- **DELETE /api/products/12** → elimina il prodotto

```

# models.py
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=6, decimal_places=2)

# serializers.py
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'

# views.py
class ProductViewSet(viewsets.ModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer

# urls.py
router = DefaultRouter()
router.register(r'products', ProductViewSet)
urlpatterns = router.urls

```

Figura 8: esempio Django

2.6 – GraphQL

GraphQL è un linguaggio di interrogazione per API sviluppato da Facebook nel 2012 e reso open-source nel 2015. È stato progettato per superare alcune limitazioni strutturali delle REST API, in particolare la rigidità nel recupero dei dati. A differenza del modello REST, dove ogni endpoint rappresenta una risorsa o un'azione predefinita, GraphQL espone un *singolo endpoint* (di norma `/graphql`) che consente al client di costruire in modo flessibile la struttura della risposta desiderata, specificando con precisione quali campi ottenere e con quale livello di annidamento.

Questo paradigma, definito come client-driven data fetching, ribalta l'approccio tradizionale: non è più il server a determinare la forma dei dati restituiti, ma è il client a definire ciò che serve, minimizzando sia l'over-fetching (recupero di dati superflui) che l'under-fetching (dati insufficienti che obbligano a ulteriori chiamate). Ciò consente una comunicazione più efficiente tra front-end e back-end, soprattutto in contesti complessi come dashboard amministrative o interfacce multicanale.

Uno degli aspetti più potenti di GraphQL è la sua capacità di esprimere relazioni nidificate tra le entità. In un'unica query, è possibile richiedere, ad esempio, una lista di prodotti, con all'interno le relative varianti, le disponibilità nel magazzino e le informazioni sul fornitore. Questo elimina la necessità di orchestrare numerose chiamate REST sequenziali o parallele, semplificando il codice e riducendo la latenza percepita.

Nel contesto del mio progetto, GraphQL è stato utilizzato in particolare per l'integrazione con le API di Shopify, una delle piattaforme e-commerce più adottate a livello globale. Shopify espone entrambe le interfacce REST e GraphQL, ma consiglia fortemente l'utilizzo di quest'ultima per la maggior parte dei casi d'uso, specialmente in contesti ad alto volume. La motivazione principale risiede nella gestione del *rate limiting*: mentre le API REST sono soggette a un numero fisso di richieste al minuto, le API GraphQL di Shopify utilizzano un sistema di crediti per unità di complessità, che assegna un "costo" differente a ogni query in base alla quantità e profondità dei dati richiesti. Questo modello, più sofisticato, premia le richieste ottimizzate e riduce i blocchi forzati, migliorando l'efficienza complessiva.

GraphQL fornisce inoltre un meccanismo formale di introspezione dello schema: è possibile interrogare il server per ottenere la lista dei tipi, campi, descrizioni e relazioni disponibili. Questo rende

l'API auto-documentata e facilita l'integrazione da parte di altri sviluppatori, senza necessità di consultare manuali esterni.

Shopify aggiorna costantemente il proprio schema GraphQL per supportare nuove funzionalità e best practice. Il loro schema pubblico è organizzato secondo logiche chiare e scalabili, con un tipo QueryRoot principale che espone risorse come products, orders, customers, e un tipo MutationRoot per modificare lo stato del sistema (es. creare un ordine, aggiornare una giacenza). Ogni campo è corredato di metadata e descritto in dettaglio, rendendo lo sviluppo più sicuro e prevedibile.

Secondo la Developer Survey 2024 di Stack Overflow, il 24% degli sviluppatori back-end ha utilizzato GraphQL nei propri progetti, e oltre il 90% ha espresso un giudizio positivo sulla sua esperienza d'uso. In particolare, GraphQL risulta preferito nei progetti moderni con interfacce complesse (SPA, mobile app, dashboard interattive), dove la flessibilità nella selezione dei dati è un requisito critico. Shopify, che ha implementato il proprio schema pubblico già dal 2018, lo utilizza oggi come API principale per tutte le soluzioni headless e per le integrazioni avanzate tramite app personalizzate.

In media, l'utilizzo di GraphQL ha permesso di ridurre del 40–50% il numero di chiamate necessarie per sincronizzare grandi cataloghi prodotti rispetto a un approccio REST tradizionale, con un impatto diretto sul tempo di risposta dell'applicazione e sul carico dei server.

In conclusione, GraphQL ha rappresentato un elemento chiave nell'interoperabilità tra il sistema gestionale interno e la piattaforma Shopify. La sua flessibilità, unita alla profondità semantica delle query e a una gestione più efficiente delle risorse, ha permesso di sviluppare integrazioni robuste, ottimizzate e più facili da mantenere nel tempo.

2.6.1 – Esempio di query GraphQL vs REST

Richiesta REST: i titoli dei primi 5 prodotti

- GET /products
- GET /products/{id}

```
{
  products(first: 5) {
    edges {
      node {
        id
        title
        variants(first: 3) {
          edges {
            node {
              price
              inventoryQuantity
            }
          }
        }
      }
    }
  }
}
```

Figura 9: esempio richiesta GraphQL

In questo esempio, con una sola chiamata è possibile ottenere i titoli dei prodotti, le varianti, i prezzi e le giacenze, evitando round trip multipli come avviene con REST. Questo approccio ha migliorato sensibilmente le prestazioni di sincronizzazione del mio sistema gestionale, riducendo il tempo medio di risposta delle operazioni batch su Shopify.

3

Cap. 3

Funzionalità e Finalità del progetto

Questo capitolo descrive nel dettaglio le funzionalità sviluppate nel sistema gestionale, evidenziandone gli obiettivi pratici e il valore aggiunto per l'attività quotidiana della PMI. Ogni funzionalità è stata progettata per rispondere a esigenze reali emerse durante la fase di analisi, con l'obiettivo di digitalizzare i flussi operativi e ridurre il margine di errore nella gestione manuale.

Si parte con l'analisi dei requisiti funzionali, che ha guidato la progettazione dell'interfaccia e dei processi principali. Successivamente vengono analizzate le funzionalità legate alla gestione del magazzino, dagli ordini ai fornitori al carico delle giacenze e alla tracciabilità delle varianti.

Il capitolo affronta poi la gestione del flusso di cassa e la contabilità, con rappresentazione visiva tramite grafici e dashboard interattive. Viene descritta anche la gestione delle informazioni per il web, inclusi i media dei prodotti, i metadati e la struttura delle immagini, per garantire una presenza online coerente e aggiornata. Infine, si analizza la parte relativa alla sincronizzazione dei canali di vendita.

3.1 - Analisi dei requisiti e casi d'uso

Il negozio per cui è stato sviluppato il sistema opera nel settore dell'abbigliamento e degli articoli militari, con un'ampia gamma di prodotti che spaziano da vestiti formali a oggettistica storica. L'attività è dotata di una sede fisica con relativo magazzino, ma negli ultimi anni ha esteso le proprie operazioni anche all'online, attraverso Shopify, Amazon ed eBay.

In precedenza, la gestione era affidata a un sistema semi-manuale, risalente a oltre vent'anni fa, che si basava su software obsoleti e processi ripetitivi, con evidenti limiti in termini di affidabilità, scalabilità e sicurezza dei dati. Questo portava a problemi frequenti come duplicazioni di prodotti, confusione tra barcode e aggiornamenti errati delle scorte.

Il nuovo sistema nasce con l'obiettivo di integrare tutte le operazioni, migliorare la qualità del dato e fornire una piattaforma moderna, sicura e accessibile che integrasse l'intero flusso.

3.1.1 - Requisiti funzionali principali

Tabella 1: Analisi dei Requisiti

MAGAZZINO	Creazione/modifica di ordini a fornitore, registrazione delle consegne parziali o complete, gestione delle varianti, controllo automatico delle giacenze.
PRODOTTI	Visualizzazione, ricerca e modifica dei prodotti; aggiornamento delle varianti, delle immagini.
CASSA	Inserimento prodotti tramite barcode, gestione resi, buoni, sospesi e sconti; selezione cliente, tipo di pagamento e stampa scontrino.
CONTABILITÀ	Calcolo automatico di guadagni, margini e acquisti; visualizzazione tramite dashboard con grafici e tabelle.
UTENTI	Accesso differenziato per reparto; autenticazione sicura; accesso limitato alle funzioni in base all'account.
CANALI ONLINE	Pubblicazione e sincronizzazione prodotti su Shopify, Amazon ed eBay; gestione immagini e metadati; aggiornamento automatico delle giacenze.

3.1.2 - Requisiti non funzionali

- Il sistema funziona su più postazioni collegate alla rete interna del negozio.
- Le operazioni devono aggiornarsi in tempo reale per tutti gli utenti attivi.
- Deve essere garantita la sicurezza dei dati tramite autenticazione e backup automatici.
- Deve essere possibile utilizzare più stampanti (es. scontrini, etichette) in modo integrato.
- L'interfaccia deve essere semplice e accessibile anche da personale non tecnico.

3.1.3 - Casi d'uso principali

Vendita al dettaglio (cassa)

1. Il cliente porta i prodotti in cassa.
2. L'operatore effettua il login se non ancora effettuato.
3. Tramite il lettore barcode vengono inseriti i prodotti.
4. L'operatore può applicare sconti, aggiungere buoni, modificare quantità o prezzi.

5. Si seleziona il metodo di pagamento.
6. Il sistema registra il conto, aggiorna le giacenze e stampa lo scontrino.

Gestione ordini a fornitore

1. L'utente accede alla sezione ordini.
2. Crea un nuovo ordine specificando fornitore, data prevista e nota.
3. Aggiunge uno o più prodotti, definendo quantità e prezzo.
4. Quando arriva la merce, inserisce una o più consegne con le varianti ricevute.
5. Avviene l'aggiornamento automatico delle scorte.
6. L'utente può cambiare lo stato dell'ordine (ricevuto, parzialmente ricevuto, annullato, ecc.).

Gestione sincronizzazione e pubblicazione online

1. L'utente completa tutte le informazioni sul prodotto locale.
2. Sceglie il canale di destinazione (Shopify, eBay).
3. Inserisce i dati richiesti dal marketplace (es. collections, tag, descrizione HTML, tempi di spedizione).
4. Ho sviluppato il caricamento tramite API e aggiorna lo stato del prodotto.
5. Settimanalmente, task automatiche aggiornano qty, prezzi e immagini.

3.1.4 - Considerazioni finali

L'analisi dei requisiti ha permesso di progettare un sistema in grado di rispondere concretamente alle esigenze della PMI. Le funzionalità coprono tutte le aree operative, garantendo un alto livello di automazione, riduzione degli errori e controllo integrato. Nei prossimi sottocapitoli saranno approfondite le singole componenti del sistema seguendo il flusso logico dell'interfaccia utente e il comportamento del back-end.

3.2 - Gestione del magazzino

La gestione del magazzino rappresenta il cuore operativo dell'intero sistema gestionale, poiché da essa dipende l'integrità e l'allineamento di tutti i flussi legati a vendite, acquisti, sincronizzazione online e controllo contabile.

Un dato aggiornato e unificato consente non solo di evitare errori operativi, ma anche di prendere decisioni migliori in termini di riordino, rotazione degli articoli e promozioni. Per questo motivo, la progettazione del modulo di magazzino è stata affrontata con particolare attenzione alla coerenza interna dei dati, all'automazione dei processi e all'interazione fluida tra front-end e back-end.



Figura 10: Diagramma a Flusso per aggiungere ordini al magazzino

3.2.1 - Flusso operativo lato front-end

Nel sistema, il personale autorizzato può gestire le operazioni di magazzino accedendo all'interfaccia di amministrazione tramite browser, dopo aver effettuato l'autenticazione. Le due operazioni principali previste sono la creazione di un nuovo ordine a fornitore e l'aggiunta di prodotti e carichi a un ordine esistente.

Creazione di un nuovo ordine:

1. L'utente apre la sezione *Ordini a fornitore*.
2. Cliccando su *Nuovo ordine*, inserisce i dati
3. Dopo la creazione, l'ordine risulta in stato *aperto* e può essere modificato successivamente.

The screenshot shows the 'Gestione Ordini' (Order Management) interface. On the left is a sidebar with filters for Fornitori, Anno, Stato Ordine, Bolla, and Fattura. The main area displays a table of orders with columns for Stato, Numero Ordine, Fornitore, Data Inserimento, Data Consegna, Acquistata, Consegnata, Respinta, Dispersa, Totale Ordine, and Azioni. The table lists 9 orders with various statuses like 'Aperto', 'Parzialmente ricevuto', 'Ricevuto', and 'Annullato'. A pagination bar at the bottom indicates 'Visualizza 10 di 10 Risultati'.

Stato	Numero Ordine	Fornitore	Data Inserimento	Data Consegna	Acquistata	Consegna	Respinta	Dispersa	Totale Ordine	Azioni
Aperto	ORD-2000	Azienda 2	15/02/2025	14/03/2025	2	1	1	0	€9,94	...
Sospeso	ORD-2001	Azienda 4	17/07/2024	06/08/2024	10	10	0	0	€75,38	...
Parzialmente ricevuto	ORD-2002	Azienda 8	28/03/2025	13/04/2025	11	10	1	0	€98,65	...
Aperto	ORD-2003	Azienda 8	01/03/2025	29/03/2025	2	1	1	0	€46,18	...
Aperto	ORD-2004	Azienda 8	23/08/2024	04/09/2024	9	9	0	0	€68,89	...
Ricevuto	ORD-2005	Azienda 8	11/08/2024	08/09/2024	3	3	0	0	€46,18	...
Ricevuto	ORD-2006	Azienda 9	08/02/2025	04/03/2025	1	1	0	0	€22,44	...
Aperto	ORD-2007	Azienda 0	25/01/2025	04/02/2025	8	8	0	0	€87,19	...
Aperto	ORD-2008	Azienda 3	02/11/2024	27/11/2024	4	4	0	0	€28,65	...
Annullato	ORD-2009	Azienda 1	09/11/2024	10/11/2024	2	2	0	0	€41,91	...

Figura 11: pagina gestione ordini

Aggiunta di prodotti e carichi a un ordine esistente:

1. Dalla lista ordini, l'utente seleziona quello appena creato o già esistente (in stato *aperto* o *parzialmente ricevuto*).
2. Gli si apre la pagina dettagli dell'ordine dove può aggiungere prodotti già presenti nel sistema o crearne di nuovi direttamente.
3. Per ogni prodotto inserito, l'utente specifica: prezzo di acquisto e quantità ordinata
4. Quando il fornitore consegna la merce (anche in più tranches), l'utente apre la sezione *Carichi* dell'ordine.
5. In questa sezione, seleziona per ogni prodotto le varianti ricevute (es. 3 taglie L rosse e 5 taglie M blu) e conferma la consegna.
6. Il sistema registra la consegna come movimento di carico, aggiorna le quantità delle varianti e prodotti in tempo reale e modifica lo stato dell'ordine:
 - o *parzialmente ricevuto* se mancano ancora articoli
 - o *ricevuto* se è stato completato
 - o *sospeso oppure annullato in caso di modifiche manuali*

DETTAGLIO ORDINE									
Ordini > Dettaglio Ordine									
Informazioni Ordine									
<div> <div>Numero Ordine: ORD-2004</div> <div>Fornitore: Azienda 8</div> <div>Totale Ordine: €68,89</div> </div> <div> <div>Data Inserimento: 23/08/2024</div> <div>Data Consegna Prevista: 04/09/2024</div> <div>Quantità Totale: 9</div> </div> <div> <div>Stato Ordine: Aperto</div> </div>									
Prodotti Ordinati									
Barcode	% IVA	Prezzo con Imposta	Totale Imposta	Acquistata	Consegnata	Respinta	Dispersa	Azioni	Prezzo Acquisito
200019	€22,00	€46,18	€8,33	3	3	0	0	+ - i	€37,85
200015	€22,00	€22,72	€4,10	6	6	0	0	+ - i	€18,62

Figura 12: pagina dettaglio di un ordine

3.2.2 - Struttura dati e logica lato back-end

La struttura dati alla base del magazzino è pensata per garantire precisione, tracciabilità e aggiornamento in tempo reale delle quantità. Ogni prodotto è collegato a una o più varianti, caratterizzate da attributi come taglia e colore. Ogni variante ha un codice SKU univoco e mantiene diversi tipi di quantità: disponibile, ordinata, venduta, in carico o da correggere.

Gli ordini a fornitore sono composti da righe di prodotto e, a loro volta, da righe di variante, per gestire in modo granulare le quantità di ogni combinazione. Ogni consegna registrata viene tracciata e aggiorna direttamente le quantità disponibili, consentendo al sistema di sapere con precisione cosa è stato ricevuto e cosa è ancora in attesa.

Il sistema tratta le varianti come entità distinte, garantendo un controllo molto dettagliato sulle scorte. Le varianti tengono traccia in tempo reale di tutte le modifiche di stato, incluse consegne, vendite, resi o rettifiche manuali. Questo approccio consente di sapere, in ogni momento, la disponibilità reale di un articolo specifico (es. "jeans nero, size L") evitando errori di sovrapposizione o stock non sincronizzati.

3.2.3 - Automazioni e sicurezza del dato

Per garantire l'affidabilità del sistema nel tempo, sono previste **operazioni automatiche** (task schedulati) che vengono eseguite settimanalmente. Questi task si occupano di:

- Ricalcolare le quantità di tutte le varianti, partendo dai movimenti reali (ordini, consegne, vendite, resi).
- Sincronizzare le giacenze con i canali online (Shopify, eBay).
- Allineare le quantità tra prodotti locali e quelli pubblicati.

Ogni operazione viene registrata in modo dettagliato: ogni modifica alle quantità genera un movimento tracciato, con tipo di operazione, timestamp e utente responsabile. La struttura è pensata per garantire integrità e sicurezza: nessuna quantità viene alterata direttamente, ma sempre attraverso operazioni strutturate e autorizzate.

3.2.4 - Integrazione con gli altri moduli

Il modulo di magazzino non lavora in modo isolato, ma è profondamente connesso con tutti gli altri componenti del sistema:

- **Cassa:** ogni vendita o reso modifica le quantità a magazzino, per questo il modulo è aggiornato in tempo reale. L'operatore vede solo prodotti effettivamente disponibili.
- **Contabilità:** il valore del magazzino incide sul calcolo dei costi, dei margini e dell'inventario. I movimenti di carico sono fondamentali per valutare il valore delle giacenze.
- **Canali online:** La logica di pubblicazione prevede che siano visibili online solo i prodotti effettivamente disponibili a magazzino. La sincronizzazione delle quantità con Shopify, Amazon ed eBay parte dalle qty aggiornate dal magazzino.

Grazie alla sua posizione centrale, il modulo di magazzino diventa il nodo logico da cui si originano tutte le operazioni chiave: vendita, analisi, contabilità e marketing.

3.2.5 - Considerazioni finali

Il modulo di magazzino si configura come il perno centrale del sistema gestionale. La sua integrazione con tutti gli altri componenti (punto vendita, contabilità, vendite online) garantisce una visione completa e aggiornata dell'attività commerciale.

La struttura a varianti consente un controllo dettagliato delle scorte, anche in presenza di prodotti complessi con molte combinazioni. L'automazione dei processi riduce drasticamente il rischio di errore umano, mentre la tracciabilità completa permette di risalire a ogni operazione effettuata, offrendo trasparenza e controllo.

In un contesto in cui l'accuratezza dell'inventario è fondamentale per evitare perdite economiche e disservizi al cliente, questa architettura permette di operare con sicurezza, affidabilità e scalabilità.

3.3 - Gestione del flusso di cassa

Il modulo cassa rappresenta uno degli elementi più delicati e dinamici dell'intero sistema gestionale. Si tratta del punto in cui avviene l'interazione diretta tra l'operatore e il cliente finale, e per questo motivo è stato progettato con particolare attenzione alla semplicità d'uso, velocità operativa e integrazione automatica con le altre sezioni del sistema (magazzino, contabilità, resi, buoni, clienti).

La piattaforma consente all'operatore di eseguire tutte le operazioni quotidiane tipiche di un'attività commerciale, attraverso un'interfaccia reattiva e ottimizzata per il lettore di codici a barre. L'utente accede alla sezione cassa e può iniziare una nuova sessione di vendita in qualsiasi momento, inserendo rapidamente i prodotti, applicando sconti, elaborando resi o buoni e completando il pagamento con pochi passaggi. Tutte le operazioni vengono tracciate e sincronizzate in tempo reale, aggiornando automaticamente le quantità a magazzino e i dati contabili.

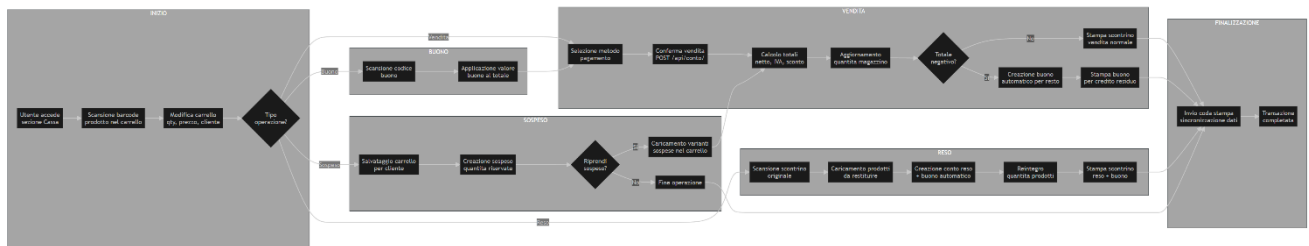


Figura 13: Diagramma Flusso di Cassa

3.3.1 - Flusso operativo completo lato front-end – vendite, resi, sospesi e buoni

Il flusso principale parte dalla pagina cassa, dove può iniziare un nuovo acquisto semplicemente scansionando il codice a barre dei prodotti. Ogni scansione aggiunge il prodotto corrispondente al carrello, identificato tramite SKU e associato alla variante specifica (taglia, colore, ecc.).

All'interno del carrello, l'operatore ha la possibilità, tramite un tastierino su schermo di:

- Modificare quantità, prezzo unitario e sconto di ciascun prodotto
- Applicare uno sconto totale
- Rimuovere prodotti singoli
- Selezionare o aggiungere un cliente all'ordine

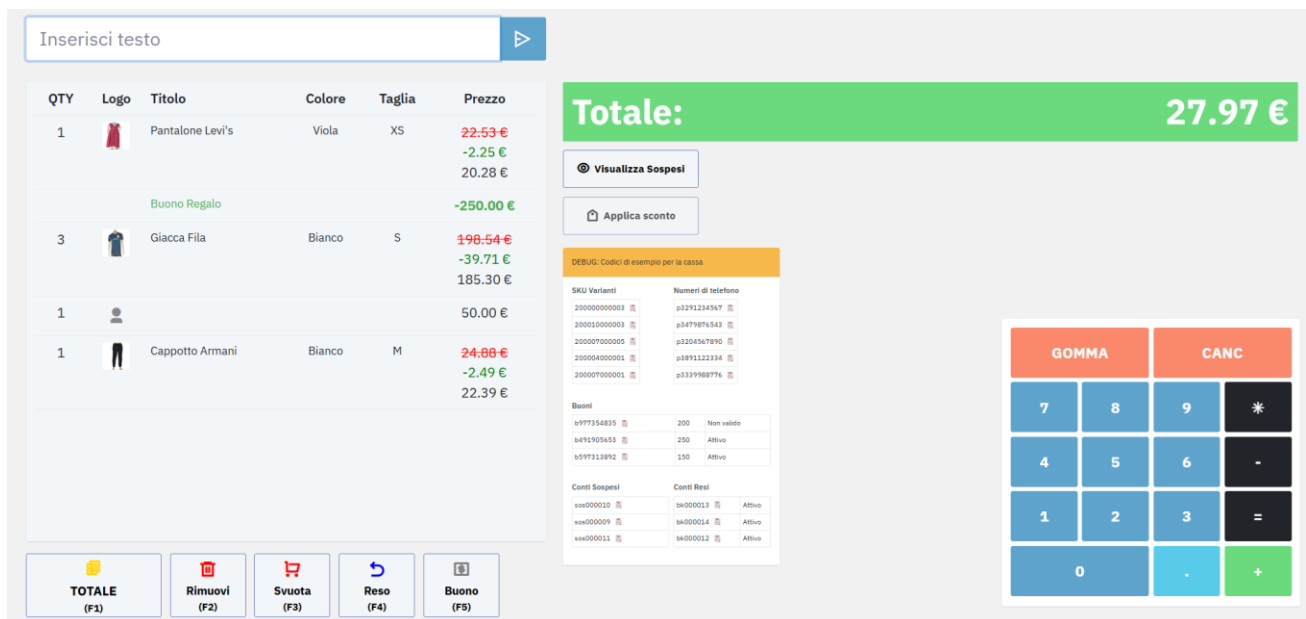


Figura 14: esempio utilizzo pagina cassa

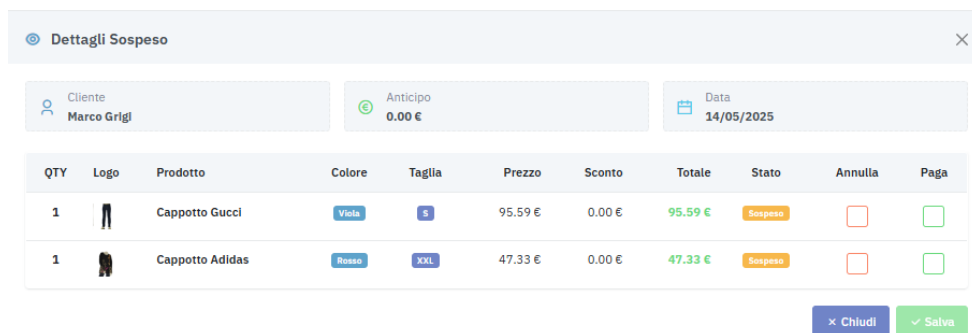
Una volta completata la compilazione del carrello, l'operatore seleziona il metodo di pagamento (contanti, carta, buono, combinazioni), e conferma la vendita. Il sistema registra la transazione come "conto", associa il cliente, aggiorna il magazzino, e stampa lo scontrino.

3.3.2 - Operazioni avanzate supportate

Il modulo supporta numerose casistiche complesse, senza uscire dal flusso della cassa:

- Resi: è possibile scansionare lo scontrino di un reso per caricare automaticamente i prodotti restituiti nel carrello. Una volta confermata l'operazione, viene generato uno scontrino di reso con relativo buono del valore corrispondente, che potrà essere utilizzato in acquisti successivi. Le quantità vengono reintegrate nel magazzino.
- Sospesi: in caso di cliente indeciso o momentaneamente impossibilitato a completare l'acquisto, è possibile salvare il carrello come scontrino sospeso, associandolo eventualmente a un cliente. In un secondo momento, lo scontrino può essere riaperto, modificato e finalizzato.
- Buoni regalo: è possibile caricare un buono tramite barcode. Il sistema ne calcola il valore e lo applica al totale dell'ordine. Se il valore del buono supera il totale, viene generato un nuovo buono residuo con il credito restante.

Queste funzionalità per garantire massima flessibilità e adattarsi a qualsiasi esigenza operativa del punto vendita. In tutte le operazioni, la fluidità dell'interfaccia e la gestione automatica delle logiche di calcolo (residui, differenze, autorizzazioni) permettono all'operatore di concentrarsi sull'esperienza del cliente.



QTY	Logo	Prodotto	Colore	Taglia	Prezzo	Sconto	Totale	Stato	Annulla	Paga
1		Cappotto Gucci	Violeta	S	95.59 €	0.00 €	95.59 €	Sospeso	<input type="checkbox"/>	<input type="checkbox"/>
1		Cappotto Adidas	Rosso	XXL	47.33 €	0.00 €	47.33 €	Sospeso	<input type="checkbox"/>	<input type="checkbox"/>

Figura 15: dettagli di un acquisto sospeso



Codice	Valore	Valore Residuo	Tipo	Data Inserimento	Stato	Azioni
617359557	200.00 €	200.00 €	Regalo	14/05/2025	Attivo	
663096713	250.00 €	250.00 €	Regalo	14/05/2025	Attivo	
597313892	150.00 €	150.00 €	Regalo	14/05/2025	Attivo	
491905653	250.00 €	250.00 €	Regalo	14/05/2025	Attivo	
977354835	200.00 €	0.00 €	Regalo	14/05/2025	Utilizzato	

Figura 16: lista dei buoni

3.3.3 - Ruolo del back-end: logica e tracciamento

Al centro del sistema si trova il concetto di Conto, che rappresenta ogni singola transazione eseguita tramite la cassa, sia essa una vendita, un reso, un buono o uno scontrino sospeso.

Ogni conto è composto da una o più righe prodotto, alle quali sono associati informazioni come quantità, prezzo unitario, eventuale sconto, metodo di pagamento e cliente (quando presente). Il servizio incaricato della gestione del conto (ContoService) si occupa di assegnare un progressivo giornaliero, calcolare automaticamente tutti i totali (imponibile, IVA, totale pagato), e salvare tutte le informazioni in modo strutturato e sicuro.

La gestione completa include anche le operazioni di reso, generando un nuovo conto con pagamento di tipo *reso*, registrando i prodotti restituiti e aggiornando le quantità a magazzino. La tracciabilità è garantita: ogni reso può essere riconciliato con il conto originario da cui è stato generato.

Le vendite sospese permettono di salvare temporaneamente un carrello, riservando i prodotti a un cliente. Il back-end gestisce lo stato del sospeso (in attesa, pagato, annullato), eventuali acconti e la variazione delle scorte. Al momento del pagamento, il sospeso può essere recuperato e trasformato in un conto definitivo.

Per quanto riguarda i buoni, il sistema crea codici univoci con valore monetario definito, validità temporale e tracciamento dello stato. I buoni possono essere riscattati parzialmente su più conti, e ogni utilizzo crea un nuovo buono del valore rimanente, se necessario. Questo garantisce un'esperienza fluida per il cliente e un controllo rigoroso per l'azienda.

Infine, la stampa degli scontrini è gestita tramite una coda di stampa intelligente che invia i documenti alle stampanti abilitate, gestendone lo stato e permettendo la ristampa in caso di necessità. Ogni tipo di documento – conto, reso, buono, sospeso – ha un suo template specifico.

Tutte le operazioni di cassa sono integrate nel sistema in modo nativo, aggiornando automaticamente le quantità del magazzino e alimentando la base dati per l'analisi contabile. In questo modo, ogni transazione diventa un nodo tracciabile all'interno dell'ecosistema gestionale, assicurando trasparenza operativa, precisione fiscale e coerenza tra front-end e back-end.

3.4 – Contabilità

Il modulo di analisi contabile del sistema gestionale ha come obiettivo la fornitura di strumenti chiari, aggiornati e accessibili per analizzare l'andamento economico dell'attività. A differenza dei software fiscali specializzati, questa sezione permette alla PMI di monitorare in tempo reale le proprie performance. Attraverso dashboard e grafici, l'utente può esplorare in modo intuitivo dati relativi a vendite, acquisti, margini di guadagno e valore dell'inventario, senza dover ricorrere a fogli di calcolo esterni o analisi manuali. L'integrazione nativa con tutti i moduli del sistema assicura coerenza e aggiornamento continuo dei dati.

3.4.1 - Visualizzazione e analisi dei conti

La pagina dei conti offre una panoramica completa di tutte le transazioni registrate nel sistema. Ogni conto rappresenta una singola operazione di vendita, reso, sospeso o buono, ed è associato a uno o più prodotti, cliente, metodo di pagamento e timestamp. L'interfaccia permette di:

- filtrare per data, tipo di operazione, canale di vendita, cliente o utente che ha registrato l'operazione;
- aprire ogni conto per esaminare le righe prodotto, sconti applicati, buoni utilizzati e totale pagato;
- esportare i risultati per elaborazioni esterne.

Questo strumento è fondamentale sia per il monitoraggio quotidiano, sia per eventuali verifiche retroattive su anomalie o contestazioni.

CONTI

Q Filtri

Cancella tutto

Giorno

Mese

Anno

Giorno

Mese

2025

Canale

Seleziona Canale

Pagamento

Seleziona Pagamento

Ricerca cliente

Nome cliente...

Applica Filtri

Lista Conti

#	Data	Totale	Sconto	Pagamento	Canale	Pagamento	Cliente	Azioni
1	08/08/2024	238.38 €	0.00 €	290.82 €	Magazzino	Assegno		...
1	11/02/2025	356.03 €	0.00 €	434.36 €	Shopify	Assegno		...
1	29/01/2025	391.93 €	0.00 €	478.16 €	Altro	Contanti		...
1	01/07/2024	476.07 €	0.00 €	580.81 €	Ebay	Carta di credito		...
1	20/03/2025	483.07 €	0.00 €	552.74 €	Shopify	Bonifico		...
1	11/11/2024	368.98 €	0.00 €	450.16 €	Magazzino	Contanti		...
1	16/11/2024	450.40 €	0.00 €	549.49 €	Ebay	Bonifico		...
1	03/07/2024	539.33 €	0.00 €	657.98 €	Ebay	Contanti		...
0	14/05/2025	0.00 €	0.00 €	0.00 €	Magazzino	Sospeso	Marco Grigi	...
0	14/05/2025	0.00 €	0.00 €	0.00 €	Magazzino	Sospeso	Mario Rossi	...
0	14/05/2025	0.00 €	0.00 €	0.00 €	Magazzino	Sospeso	Valentina Rosa	...

Figura 17: pagina dei conti

Dettagli Conto							X	
Canale Ebay		Pagamento Contanti		Data 03/07/2024				
Prezzo 539.33 €		Sconto 0.00 € 0.00 %		Pagato 657.98 €				
Quantità	Prezzo	Sconto	Totale	Titolo	Colore	Taglia		
3	29.48 €	8.00	88.44	Vestito Levi's	Aranzone	S		
2	82.90 €	2.00	165.80	Vestito Adidas	Viola	L		
3	88.33	0.00	264.99	Pantaloni Levi's	Blu	S		
3	46.25 €	1.00	138.75	Vestito Levi's	Viola	S		

Figura 18: esempio dei dettagli di un conto

3.4.2 - Analisi di fatturato, acquisti e guadagni

Le sezioni dedicate a fatturato, acquisti e guadagni condividono una struttura simile e sono pensate per offrire una visione sintetica ma profonda dell'andamento economico.

La pagina del Fatturato mostra l'insieme delle vendite effettuate in un determinato intervallo di tempo, con possibilità di aggregazione giornaliera o mensile. I dati vengono rappresentati tramite:

- grafici a linee che evidenziano l'andamento temporale;
- grafici a torta che mostrano la distribuzione per reparto e marca;
- confronto diretto tra due periodi differenti.

La sezione degli acquisti visualizza l'importo speso per gli ordini a fornitore. La struttura è analoga al fatturato, con visualizzazioni per fornitore, categoria e periodo, utili a valutare la dinamica dei costi e ottimizzare i rifornimenti.

Quella dei Guadagni, confrontando i dati di fatturato e acquisti, il sistema calcola il guadagno lordo. Le analisi si possono effettuare per giorno, mese, reparto o marca.

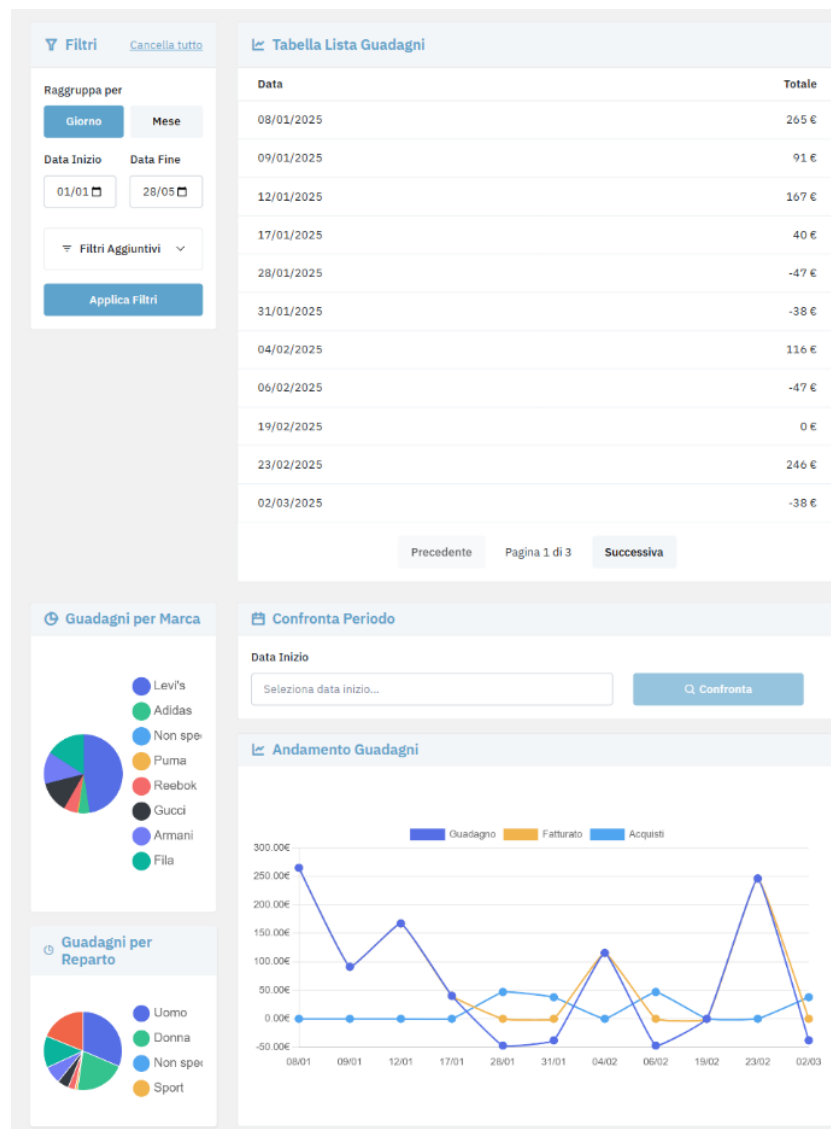


Figura 19: pagina dei Guadagni

3.4.3 - Inventario: valore e quantità delle scorte

Una pagina specifica permette di consultare lo stato dell'inventario a una determinata data. Per ogni prodotto (e per ogni sua variante), è possibile visualizzare:

- quantità disponibile in magazzino;
- costo di carico unitario;
- valore totale della giacenza ($qty \times costo$).

Questa funzione è utile per la chiusura di periodo (mensile o annuale) e il controllo del valore immobilizzato in stock;

INVENTARIO Gestione Magazzino > Inventario

Q Filtri

Data

20/05/2025

Applica Filtri

Esporta Tabella

Lista Inventario

Titolo	Barcode	Reparto	Quantità Rimanente	Prezzo Listino	Totale	Azioni
T-shirt Fila	200015	Bambina	10	26.60 €	266.00 €	Visualizza
Cappotto Gucci	200005	Bambino	5	63.47 €	317.35 €	Visualizza
Pantalone Puma	200011	Bambino	1	11.64 €	11.64 €	Visualizza
Shorts Reebok	200008	Bambino	4	34.85 €	139.40 €	Visualizza
Felpa Armani	200019	Casual	9	54.07 €	486.63 €	Visualizza
Felpa Reebok	200018	Elegante	3	36.34 €	109.02 €	Visualizza
Cappotto Levi's	200016	Outdoor	3	38.63 €	115.89 €	Visualizza
Cappotto Adidas	200017	Sport	1	26.27 €	26.27 €	Visualizza

Figura 20: pagina Inventario

3.4.4 - Back-end: generazione dei dati contabili

La generazione dei dati contabili avviene in modo completamente automatico e trasparente per l'utente finale. Il back-end si basa su una struttura a tre livelli:

1. Livello transazionale: registra tutte le operazioni (vendite, resi, acquisti, carichi) attraverso entità come Conto, RigaConto, RigaReso, ConsegneVarianteOrdine.
2. Livello di servizio: attraverso il ContoService, il sistema elabora e aggrega i dati grezzi in funzione dei parametri richiesti (periodo, reparto, marca, ecc.).
3. Livello API: espone endpoint filtrabili e ottimizzati che alimentano le dashboard del front-end.

I metodi implementati (come `get_fatturato()`, `get_guadagni()`) calcolano le metriche di vendita escludendo le voci non rilevanti (es. resi o buoni), e permettono il filtraggio dinamico per data, canale, marca, stagione.

Per il calcolo dei guadagni, il sistema mette in relazione le vendite con le consegne registrate dello stesso prodotto, permettendo una stima del margine lordo. Grazie all'uso di tecniche avanzate come le query ottimizzate con `select_related` e `annotate`. Oppure per la cache dei risultati intermedi e la gestione atomica delle transazioni.

Ho ottimizzato il sistema per garantire prestazioni elevate e coerenza dei dati, anche in presenza di migliaia di operazioni. Tutti i dati visualizzati nei grafici e nelle tabelle sono generati in tempo reale o su richiesta, e possono essere esportati o confrontati tra diversi intervalli temporali.

3.5 - Gestione dei media, dei metadati e delle informazioni per il web

Nel contesto della vendita online, la qualità dei contenuti visivi e informativi di un prodotto rappresenta un fattore critico per il successo commerciale. I clienti si aspettano immagini nitide, descrizioni dettagliate e informazioni complete: per questo, il sistema gestionale integra un modulo specifico per la gestione dei media, dei metadati e delle informazioni destinate alla pubblicazione web.

I file multimediali sono organizzati in una unità disco condivisa (Z:) accessibile al sistema e strutturata in modo logico per facilitare il caricamento, la consultazione e la riutilizzazione dei materiali. Le principali categorie in cui sono suddivisi i contenuti sono:

- **Marche:** immagini e loghi utilizzati per l'identificazione nei canali online.
- **Fornitori:** foto istituzionali o grafiche fornite dai produttori.
- **Template:** file grafici utilizzati per la stampa di scontrini, etichette o descrizioni eBay preformattate.
- **Prodotti:** una struttura più articolata che comprende:
 - *img_originali*: le fotografie ad alta risoluzione caricate manualmente;
 - *img_web*: versioni ottimizzate per la pubblicazione sui marketplace;
 - *thumb*: miniature usate per le anteprime rapide nel gestionale;
 - *img360*: cartella dedicata alle foto a 360° provenienti da sistemi automatizzati.

Una particolare attenzione è stata rivolta proprio all'integrazione con un sistema esterno di fotocamere multiposizionate, che consente la cattura simultanea di 16 scatti da diverse angolazioni del prodotto. Il sistema riceve queste immagini automaticamente tramite API, le salva nella sezione dedicata del prodotto e le collega alle inserzioni su Shopify ed eBay per permettere una visualizzazione interattiva a rotazione. Questa funzione migliora notevolmente la presentazione degli articoli, soprattutto quelli più complessi o ad alto valore, incrementando la percezione di affidabilità e la probabilità di conversione all'acquisto.

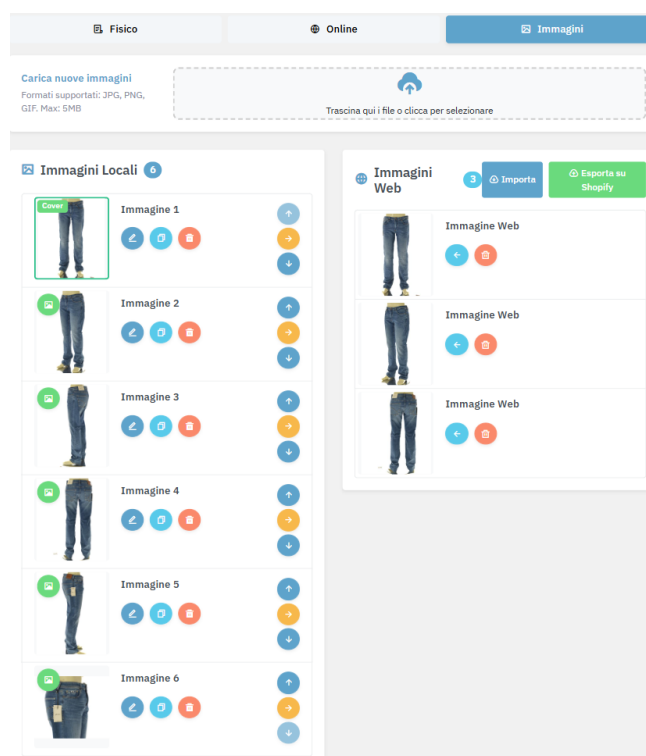


Figura 21: pagina gestione immagini prodotto

Oltre alla gestione dei contenuti visivi, il sistema offre la possibilità di modificare tutte le informazioni descrittive che andranno pubblicate online, direttamente dalla scheda di modifica del prodotto. Questo approccio consente di mantenere un'unica base dati coerente tra gestionale interno e marketplace, evitando ridondanze o errori di sincronizzazione.

Tra i campi modificabili vi sono:

- Titolo e descrizione breve/lunga, pensati per adattarsi ai diversi formati richiesti dai portali online.
- Prezzo di vendita, eventualmente differenziabile per canale (es. un prezzo per Shopify e uno per eBay).
- Categorie e collezioni, collegabili ai tag e filtri utilizzati dagli utenti online.
- Aspetti tecnici e metadati specifici, in particolare quelli richiesti da eBay, come: marca, materiale, stagione, tipo di vestibilità, genere e altre informazioni obbligatorie per garantire la validazione dell'inserzione.
- Set di immagini da associare al canale, selezionabili dall'archivio del prodotto.

Tutti questi dati vengono salvati nel sistema e resi disponibili per la successiva sincronizzazione con i canali di vendita online. La logica alla base è quella della personalizzazione coordinata: ogni prodotto può adattare la propria identità digitale alle esigenze specifiche del canale di destinazione, mantenendo però coerenza interna.

La gestione avanzata dei media e dei dati descrittivi rappresenta un tassello fondamentale per qualsiasi attività che voglia operare con successo nel commercio elettronico. Il sistema realizzato consente non solo di archiviare in modo ordinato e sicuro le immagini e i template associati ai prodotti, ma anche di governare attivamente l'intera esperienza visiva e informativa offerta all'utente finale.

3.6 - Pubblicazione e sincronizzazione con i canali di vendita web

La gestione della pubblicazione dei prodotti online è uno degli aspetti più delicati e strategici per un'attività che opera su più canali digitali. Ogni piattaforma presenta specifici requisiti, formati, flussi e vincoli di validazione. Per questo motivo, il sistema sviluppato offre un modulo dedicato alla pubblicazione e sincronizzazione, che consente di gestire in modo coordinato tutte le operazioni, riducendo al minimo il rischio di errori e rendendo il processo più efficiente.

Ogni prodotto creato localmente nel gestionale possiede una sezione "Online", accessibile dalla sua scheda di dettaglio. Da lì, l'operatore può completare tutti i campi necessari alla pubblicazione e decidere su quali canali vendere. Il flusso è stato pensato in maniera sequenziale, iniziando dalla pubblicazione su Shopify, per poi procedere su eBay e infine, tramite collegamento, su Amazon.

3.6.1 - Pubblicazione su Shopify

Shopify rappresenta il primo punto di pubblicazione, anche perché funge da ponte verso Amazon. Una volta inserito il prodotto a livello locale, l'operatore può accedere alla sezione dedicata e compilare le seguenti informazioni:

- Collections: ottenute dinamicamente tramite API dal back-end, che a sua volta le recupera direttamente dallo store Shopify. Viene fornita all'utente l'intera struttura ad albero delle collezioni, semplificando la selezione di massimo due collections.
- Tag: utili per il filtraggio e la categorizzazione degli articoli.
- Prezzo di vendita: eventualmente diverso da quello del gestionale e una percentuale di aggiunta.

Una volta compilati i campi, è possibile lanciare la pubblicazione automatica tramite API. La creazione del collegamento avviene tramite diretto tra il prodotto locale e l'ID Shopify generato, consentendo così futuri aggiornamenti o rimozioni in modo sincronizzato e controllato. Ogni volta che viene aperta la pagina viene fatta una sincronizzazione col prodotto web, viene chiesto in caso di differenze quale versione si vuole mantenere e di conseguenza vengono fatti gli aggiornamenti.

Figura 22: dettagli Shopify di un prodotto

3.6.2 - Pubblicazione su eBay

Dopo la pubblicazione su Shopify, il prodotto può essere predisposto per l'inserimento su eBay. La procedura è più articolata, poiché richiede:

- Categoria di vendita: il sistema propone un campo di ricerca testuale per facilitare l'inserimento della categoria. A ogni carattere digitato, il back-end interroga le API eBay e mostra un menu di autocompletamento con le corrispondenze esatte, inclusa la gerarchia (es. *Uomo > Giacche*).
- Una volta selezionata la categoria, viene restituita una tabella con tutti gli "aspects" disponibili per quel ramo. Questi includono dati tecnici e descrittivi come: marca, materiale, tessuto, tipo di prodotto, fit, genere, taglia, e molti altri obbligatori e opzionali.
- Titolo e sottotitolo: ottimizzati per l'indicizzazione interna del marketplace.
- Descrizione HTML: editabile tramite template o campo libero.
- Spedizione: tempi di consegna, costi, modalità di invio.

La pubblicazione avviene tramite API e viene salvata nel sistema l'associazione tra il prodotto locale e l'ID eBay. In caso di modifica futura, il sistema è in grado di aggiornare l'inserzione senza doverla ricreare. Ogni volta che viene aperta la pagina viene fatta una sincronizzazione col prodotto web, viene chiesto in caso di differenze quale versione si vuole mantenere e di conseguenza vengono fatti gli aggiornamenti.

Titolo eBay 80 caratteri rimanenti

Vestito Levi's

Sottotitolo eBay

Sottotitolo eBay

Categoria

Riviste / Libri e riviste / Sport Cerca

Prezzo eBay 21,11 **Percentuale Aggiunta eBay** 110

Spedizione Standard 0 **Spedizione Espressa** 0

Durata Giorni Select... **Last eBay ID** Last eBay ID

Descrizione Visualizza Anteprima

Sans Serif Normal B I U S Visualizza Anteprima

Un classico rivisitato in chiave contemporanea.

Aspetti Ebay

Nome	Richiesto	Tipo	Valori
Soggetto	Opzionale	Singolo Testo libero	Inserisci valore... ► Valori suggeriti (15)
Personalizzato	Opzionale	Singolo Solo selezione	Seleziona... ▼
Autografato	Opzionale	Singolo Solo selezione	Seleziona... ▼
Autore	Opzionale	Singolo Testo libero	Inserisci valore... ► Valori suggeriti (0)
Editore	Opzionale	Singolo Testo libero	Inserisci valore... ► Valori suggeriti (0)
Nome della pubblicazione	Opzionale	Singolo Testo libero	Inserisci valore...

Figura 23: dettagli prodotto eBay

3.6.3 - Sincronizzazione con Amazon

Per Amazon, Ho scelto di non implementare una pubblicazione diretta, ma sfrutta la sincronizzazione nativa con Shopify. Quando un prodotto viene pubblicato e attivato su Shopify, e se collegato all'account Amazon Seller Central, questo viene automaticamente replicato e gestito anche su Amazon. Questa scelta riduce il numero di interfacce e semplifica il lavoro, centralizzando le modifiche (descrizioni, immagini, prezzi) attraverso Shopify, che funge così da hub per entrambi i marketplace.

3.6.4 - Task automatici di sincronizzazione e aggiornamento

Una volta pubblicati, i prodotti restano collegati ai relativi ID esterni su Shopify, eBay e Amazon. Ho programmato l'esecuzione settimanale un task automatico di risincronizzazione completa, che:

- aggiorna tutte le quantità disponibili (qty) sincronizzandole con le giacenze locali;
- controlla e, se necessario, aggiorna immagini, descrizioni, prezzi e aspects;
- rispetta i tempi tecnici e i limiti di chiamata previsti da ogni piattaforma (come rate limit su Shopify o crediti per chiamate su eBay).

In questo modo, anche in caso di modifiche manuali o disallineamenti temporanei, il sistema garantisce una coerenza continua e automatica tra il gestionale e i canali di vendita esterni.

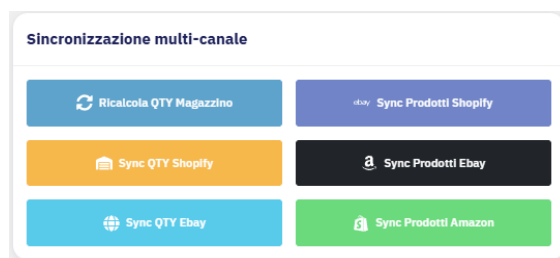


Figura 24: opzioni di task automatici da avviare

4

Cap. 4

Struttura del back-end

Il cuore del sistema gestionale è costituito da un back-end solido, modulare e scalabile, sviluppato interamente in Django e strutturato per supportare sia le funzionalità interne del negozio fisico sia l'integrazione con i principali canali di vendita online. Questo capitolo descrive in dettaglio l'architettura del back-end, partendo dalla progettazione del database, passando per la suddivisione delle API locali e web, fino ad arrivare alla gestione delle automazioni, dei webhook e della sicurezza.

Ogni componente è stato progettato per essere indipendente ma interoperabile, in modo da garantire massima flessibilità nella gestione del magazzino, nella sincronizzazione dei dati con le piattaforme esterne (Shopify, eBay, Amazon) e nel supporto ai flussi operativi quotidiani.

Nei prossimi paragrafi verranno illustrate le scelte progettuali adottate, le logiche di comunicazione tra servizi e le soluzioni implementate per garantire un'integrazione completa e centralizzata di tutte le attività della PMI.

4.1 - Struttura del database

La progettazione del database segue un approccio relazionale che centralizza tutte le informazioni necessarie alla gestione del negozio. Ho utilizzato PostgreSQL come motore, sfruttando l'ORM di Django per definire le relazioni tra le entità in modo dichiarativo e manutenibile. Il sistema si articola attorno all'entità Product, che rappresenta l'articolo base con informazioni generali come barcode, titolo, descrizioni e prezzi. Ogni prodotto è collegato a ProductVariant, che gestisce le declinazioni specifiche per taglia (VariantSize) e colore (VariantColor). Questa struttura permette di gestire un catalogo complesso dove ogni capo può avere decine di combinazioni, mantenendo per ciascuna variante un SKU univoco e quantità specifiche.

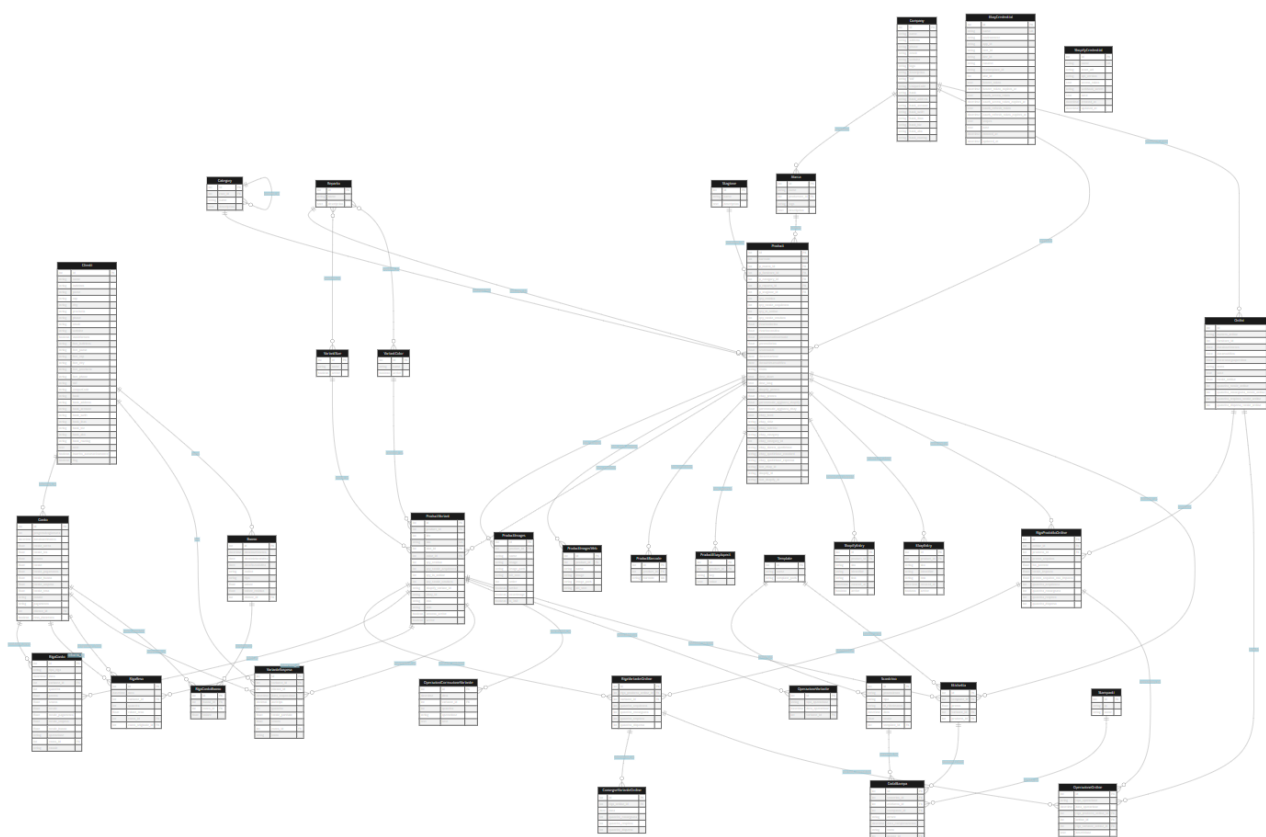


Figura 25: UML database

Le informazioni descrittive sono organizzate attraverso entità di classificazione: Category gestisce la gerarchia delle categorie con supporto per sottocategorie, Reparto identifica le aree del negozio, Stagione classifica gli articoli per periodo, mentre Marca collega ogni prodotto al relativo produttore (Company). Quest'ultima entità serve anche per gestire i fornitori, centralizzando dati fiscali, bancari e di contatto.

Per i contenuti multimediali, ProductImages archivia le immagini principali con informazioni su ordinamento e tipo, mentre ProductImagesWeb gestisce versioni ottimizzate per i canali online. La tabella ProductBarcode supporta codici aggiuntivi per prodotti con più identificativi.

La gestione degli ordini ai fornitori utilizza una struttura gerarchica: Ordini contiene le informazioni generali, RigaProdottoOrdine dettaglia i prodotti ordinati con prezzi e quantità, mentre

RigaVarianteOrdine specifica le singole varianti. ConsegneVarianteOrdine registra ogni consegna ricevuta, permettendo la gestione di arrivi parziali e il controllo delle giacenze.

Il flusso di cassa è gestito attraverso Conto, che rappresenta ogni transazione effettuata. Ogni conto è composto da righe (RigaConto) che collegano le varianti vendute ai relativi prezzi e modalità di pagamento. L'entità Clienti raccoglie i dati anagrafici e fiscali dei clienti, con campi specifici per la fatturazione quando necessaria. Per operazioni speciali, RigaReso gestisce i prodotti restituiti collegandoli al conto originale, VarianteSospesa mantiene i prodotti temporaneamente riservati, mentre il sistema dei buoni si articola tra Buono per la definizione del voucher e RigaContoBuono per il suo utilizzo nelle transazioni. OperazioniCorrezioneVariante permette modifiche manuali alle quantità quando necessario. L'integrazione con i canali online è supportata da ShopifyEntry e EbayEntry, che mantengono il collegamento tra prodotti locali e ID esterni.

Le credenziali per l'accesso alle piattaforme sono gestite attraverso ShopifyCredential e EbayCredential, con supporto per ambienti multipli e gestione automatica del rinnovo dei token. Ho basato il sistema di stampa su Scontrino ed Etichetta per definire i documenti, Template per i layout di stampa, Stampanti per la configurazione hardware e CodaStampa per gestire la sequenza di stampa con controllo degli stati e gestione degli errori. Infine, OperazioneOrdine e OperazioneVariante registrano le modifiche significative per mantenere uno storico delle operazioni, essenziale per il controllo e la verifica delle attività svolte.

4.2 - Struttura delle api locali

L'architettura delle API locali del modulo warehouse segue il pattern Model-View-Serializer tipico di Django REST Framework, organizzato secondo una divisione funzionale che rispecchia i domini operativi del sistema gestionale. L'intera struttura è coordinata da un router centrale che espone 64 endpoint differenziati per responsabilità e contesto d'uso.

La suddivisione principale si articola in quattro aree funzionali distinte: assets, product, order e cassa. Ogni area mantiene una propria organizzazione interna con views, serializers e services dedicati, garantendo separazione delle responsabilità e facilitando la manutenzione del codice. Il router registra tutti gli endpoint utilizzando il prefisso semantico che identifica immediatamente la funzione.

Il gruppo assets gestisce le entità di base del sistema attraverso ViewSet dedicati per Company, Category, Reparto, Stagione, Marca, VariantSize e VariantColor. Questi endpoint supportano operazioni CRUD complete e forniscono i dati necessari per popolare i menu di selezione nell'interfaccia utente. L'endpoint users permette la gestione degli account, mentre auth centralizza l'autenticazione e la gestione dei token.

Per la gestione dei prodotti è presente l'area product che espone ProductViewSet e ProductVariantViewSet. ProductViewSet gestisce sia la creazione che la modifica dei prodotti base, con metodi custom per la ricerca avanzata, il caricamento delle immagini e la gestione dei barcode.

ProductVariantViewSet si occupa delle singole varianti, implementando logiche complesse per il calcolo delle quantità e la sincronizzazione con i canali online.

L'area Order coordina l'intero flusso degli acquisti attraverso una gerarchia di ViewSet interconnessi. OrdiniViewSet gestisce la creazione e modifica degli ordini principali, con azioni custom per il cambio di stato e il calcolo dei totali. ConsegneVarianteOrdineViewSet registra ogni consegna ricevuta, aggiornando automaticamente le quantità tramite i quantityServices. OperazioneOrdineViewSet e OperazioneVarianteViewSet mantengono la tracciabilità delle modifiche per avere un controllo dello storico dei prodotti ordinati.

Il modulo cassa rappresenta la sezione più complessa per numero di entità gestite. ContoViewSet coordina tutte le transazioni di vendita con metodi specifici per la creazione di conti, l'applicazione di sconti e la gestione dei pagamenti multipli. ClientiViewSet gestisce l'anagrafica con ricerca per nome, email e telefono. RigaContoViewSet e RigaResoViewSet gestiscono rispettivamente le vendite e i resi, mentre VarianteSospesaViewSet coordina i prodotti temporaneamente riservati. Il sistema dei buoni è gestito da BuonoViewSet con validazione automatica delle scadenze e calcolo dei residui.

Per la stampa, ScontrinoViewSet, EtichettaViewSet e TemplateViewSet coordinano la generazione dei documenti, mentre StampantiViewSet gestisce la configurazione hardware. Ogni documento generato viene automaticamente inserito in coda attraverso l'integrazione con il sistema di stampa. La gestione unificata delle immagini è affidata alla classe ImageURLService, che standardizza la generazione degli URL per tutti i contenuti multimediali. Questo servizio gestisce sia le immagini dei prodotti, organizzate per barcode, sia i loghi di marche e aziende, garantendo percorsi coerenti e facilmente modificabili. Il servizio supporta anche la conversione di immagini Base64 provenienti dal front-end attraverso Base64ImageField.

L'endpoint home centralizza tutte le query necessarie per la dashboard principale, esponendo statistiche aggregate, timeline delle vendite e classifiche dei prodotti più venduti per canale. Questo approccio riduce il numero di chiamate multiple dal front-end e ottimizza le performance della homepage. Le funzioni di ricalcolo delle quantità sono gestite da RecalcQtyViewSet, che espone endpoint per la sincronizzazione manuale quando necessaria. I quantityServices rappresentano il livello di logica di business che coordina tutti gli aggiornamenti delle giacenze.

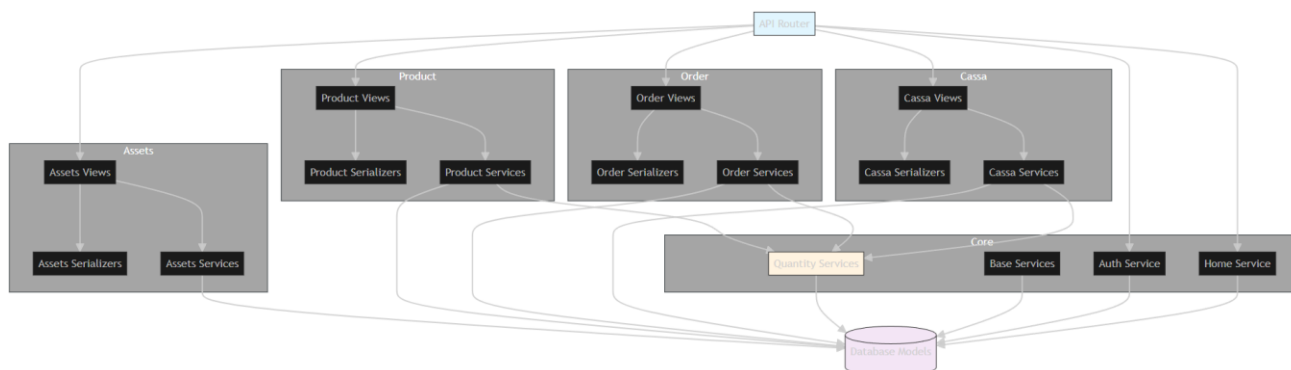


Figura 26: diagramma API Warehouse

4.3 - Struttura delle api web per lo store online (Shopify, eBay, Amazon)

Il modulo webapi coordina le integrazioni con Shopify, eBay e Amazon attraverso servizi specializzati che comunicano con le API esterne, dove i servizi gestiscono la comunicazione con le piattaforme esterne e le view espongono endpoint RESTful per il front-end. Il routing principale distribuisce le chiamate attraverso prefissi semantici che identificano immediatamente la piattaforma di destinazione.

Per Shopify, l'integrazione si basa su un'architettura a servizi specializzati che sfrutta le GraphQL API native della piattaforma. Il `product_service` coordina la creazione, modifica e sincronizzazione dei prodotti, utilizzando query GraphQL ottimizzate per ridurre il numero di chiamate necessarie. Il `collection_service` gestisce la gerarchia delle collezioni, recuperando dinamicamente la struttura ad albero per popolare i menu di selezione. L'`image_service` gestisce il caricamento e la sincronizzazione delle immagini, supportando sia l'import da Shopify verso il sistema locale che viceversa.

Il `graphql_service` centralizza tutte le chiamate GraphQL, gestendo autenticazione, rate limiting e parsing delle risposte. Questo servizio implementa retry automatici e gestione degli errori, garantendo resilienza nelle comunicazioni con l'API Shopify. Le view Shopify espongono endpoint per la gestione completa del ciclo di vita dei prodotti. `ProductShopifyView` gestisce creazione e aggiornamento, con supporto per varianti multiple e metadati complessi.

Per eBay, l'architettura si basa su `product_functions` che implementano l'intero flusso di pubblicazione attraverso le Trading API. Il sistema gestisce la complessità degli aspetti eBay, con `EbayCategoryAspectsView` che recupera dinamicamente i metadati richiesti per ogni categoria. `EbayProductCreateView` coordina la creazione completa includendo caricamento immagini su EPS, gestione varianti e mapping degli aspetti.

`EbayProductUpdateView` e `EbayProductInfoView` gestiscono rispettivamente modifiche e recupero informazioni, mentre `EbayProductSyncImagesView` sincronizza i contenuti multimediali. Il `credentials_services` per eBay gestisce la complessità dell'autenticazione OAuth e dei bearer token, con rinnovo automatico e gestione di ambienti multipli.

`EbayBearerCredentialAPIView` e `EbayOAuthCredentialAPIView` espongono endpoint per la configurazione e verifica delle credenziali, supportando sia l'ambiente sandbox che produzione.

Per Amazon, l'integrazione sfrutta la sincronizzazione nativa con Shopify attraverso Amazon Sales Channel. Il sistema implementa API dedicate per l'aggiornamento delle quantità che rispettano i feed MWS richiesti da Amazon. Gli endpoint Amazon gestiscono principalmente sincronizzazione inventario e stato ordini, delegando a Shopify la gestione dei contenuti e metadati prodotto. La gestione coordinata delle credenziali permette configurazioni multiple per ogni piattaforma, supportando account diversi per test e produzione.

Ogni servizio implementa caching intelligente per ridurre le chiamate API e rispettare i rate limit imposti dalle piattaforme. I servizi di sincronizzazione coordinano gli aggiornamenti bidirezionali, garantendo coerenza tra il sistema locale e i marketplace esterni.

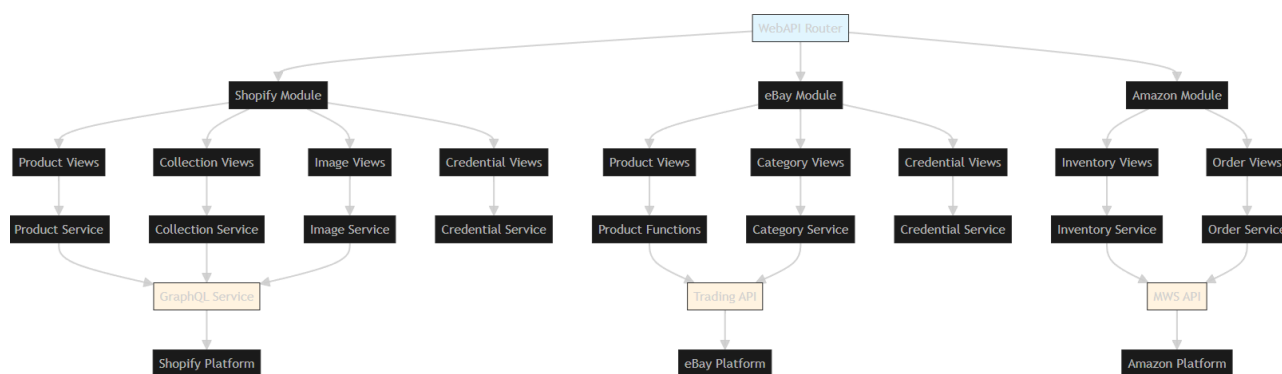


Figura 27: diagramma API Web

4.4 - Struttura delle api di stampa

Il modulo stampaapi rappresenta una componente specializzata del sistema dedicata esclusivamente alla comunicazione con le stampanti fisiche del negozio. L'architettura è volutamente semplice e focalizzata, implementando un sistema di coda che gestisce l'invio di etichette e scontrini ai dispositivi di stampa. L'organizzazione del modulo segue un pattern essenziale view-utils, dove le view espongono endpoint RESTful per la gestione della coda e le utilities si occupano della serializzazione dei dati. Il sistema gestisce due tipologie principali di documenti: etichette prodotto e scontrini di vendita, ognuna con logiche specifiche ma architettura comune.

Il sistema supporta due tipologie di etichette: quelle per scaffale, che mostrano informazioni base del prodotto, e quelle per variante specifica, che includono dettagli di taglia e colore. Per gli scontrini, l'architettura replica la stessa logica delle etichette, ma gestisce una complessità maggiore dovuta ai diversi tipi di documento. Il sistema supporta scontrini di vendita standard, buoni regalo, documenti di reso e sospensioni, ognuno con propri metadati e struttura dati specifica. Il monitoraggio dello stato avviene attraverso StatoEtichettaView e StatoScontrinoView, che permettono di verificare l'avanzamento della stampa e identificare eventuali errori. Questi endpoint implementano controlli di autorizzazione per garantire che ogni utente possa accedere solo ai propri documenti in coda. La gestione operativa è affidata a GestisciEtichettaView e GestisciScontrinoView, che espongono azioni di annullamento e ristampa. Questi endpoint permettono di gestire situazioni di errore o necessità di duplicazione, aggiornando lo stato degli elementi e registrando eventuali messaggi di errore forniti dall'utente.

Questo meccanismo garantisce la sincronizzazione tra sistema e dispositivi fisici, aggiornando automaticamente lo stato degli elementi processati. Le utilities di serializzazione gestiscono la complessità dei dati, trasformando le entità del database in strutture JSON ottimizzate per la comunicazione con le stampanti.

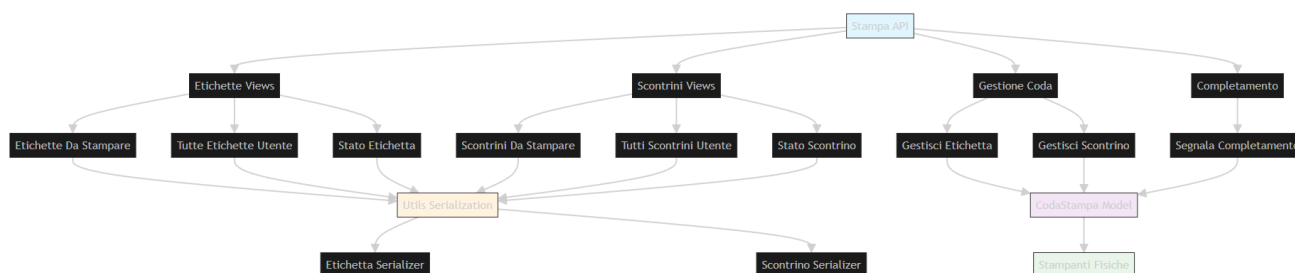


Figura 28: diagramma API Stampa

4.5 - Task automatici in background di sincronizzazione con il web

L'architettura asincrona per la gestione di operazioni lunghe e sincronizzazioni periodiche, utilizzando Celery come task queue e Redis come message broker. Celery gestisce l'esecuzione distribuita di task complessi, mentre Redis funge da message broker e storage per risultati e log di progresso.

L'architettura coordina tre categorie principali di operazioni automatiche. Il ricalcolo delle quantità analizza tutte le varianti prodotto per garantire coerenza dei dati, aggregando informazioni da acquisti, vendite, ordini e resi attraverso il QuantityService. Questo processo corregge eventuali discrepanze accumulate nel tempo e mantiene allineate le giacenze locali.

Le sincronizzazioni multicanale gestiscono l'allineamento bidirezionale con le piattaforme esterne. Task parallele coordinano per Shopify ed eBay che eseguono operazioni analoghe: sincronizzazione quantità, aggiornamento prodotti completi e gestione immagini. Per ogni piattaforma, le task recuperano dati locali, confrontano con lo stato remoto e applicano gli aggiornamenti necessari rispettando i rate limit specifici delle API. La sincronizzazione quantità coordina l'allineamento delle giacenze confrontando valori locali con quelli delle piattaforme esterne e aggiornando dove necessario. La sincronizzazione prodotti gestisce l'aggiornamento completo di metadati, varianti, collezioni e contenuti multimediali, identificando automaticamente le modifiche da applicare.

Tutte le task implementano logging dettagliato attraverso Redis, registrando progresso, errori e statistiche operative. I log vengono conservati e sono accessibili in tempo reale dal front-end per fornire feedback immediato agli utenti durante l'esecuzione. Il sistema supporta sia esecuzione manuale che automatica ogni settimana. Gli utenti possono avviare sincronizzazioni on-demand attraverso l'interfaccia web, mentre Celery Beat gestisce l'esecuzione automatica secondo schedule predefiniti.

4.6 - Hooks dalle piattaforme web

Il sistema implementa webhook per la ricezione automatica di ordini dalle piattaforme esterne, garantendo sincronizzazione immediata tra marketplace e gestionale locale. L'architettura segue un pattern comune per entrambe le piattaforme: validazione sicurezza, parsing dati, creazione entità locali e aggiornamento quantità.

La validazione della sicurezza utilizza verifica HMAC-SHA256 per Shopify e meccanismi analoghi per eBay, calcolando hash del payload e confrontandolo con gli header ricevuti. I webhook richiedono

connessioni HTTPS per garantire sicurezza delle comunicazioni, aspetto che ha richiesto implementazione di certificati SSL nel sistema come descritto in seguito. Il processing coordina operazioni atomiche: estrazione dati cliente con creazione automatica se necessario, generazione conto locale con progressivo giornaliero, creazione righe per ogni articolo collegando varianti tramite SKU, e aggiornamento immediato delle quantità residue. Il sistema registra la sorgente di ogni transazione, rendendo possibile una successiva analisi dei canali più performanti. La generazione automatica degli scontrini coordina la creazione di documenti fiscali inseriti nella coda di stampa per elaborazione immediata. L'aggiornamento delle quantità avviene in tempo reale durante il processing, decrementando giacenze e aggiornando totali prodotto per prevenire overselling.

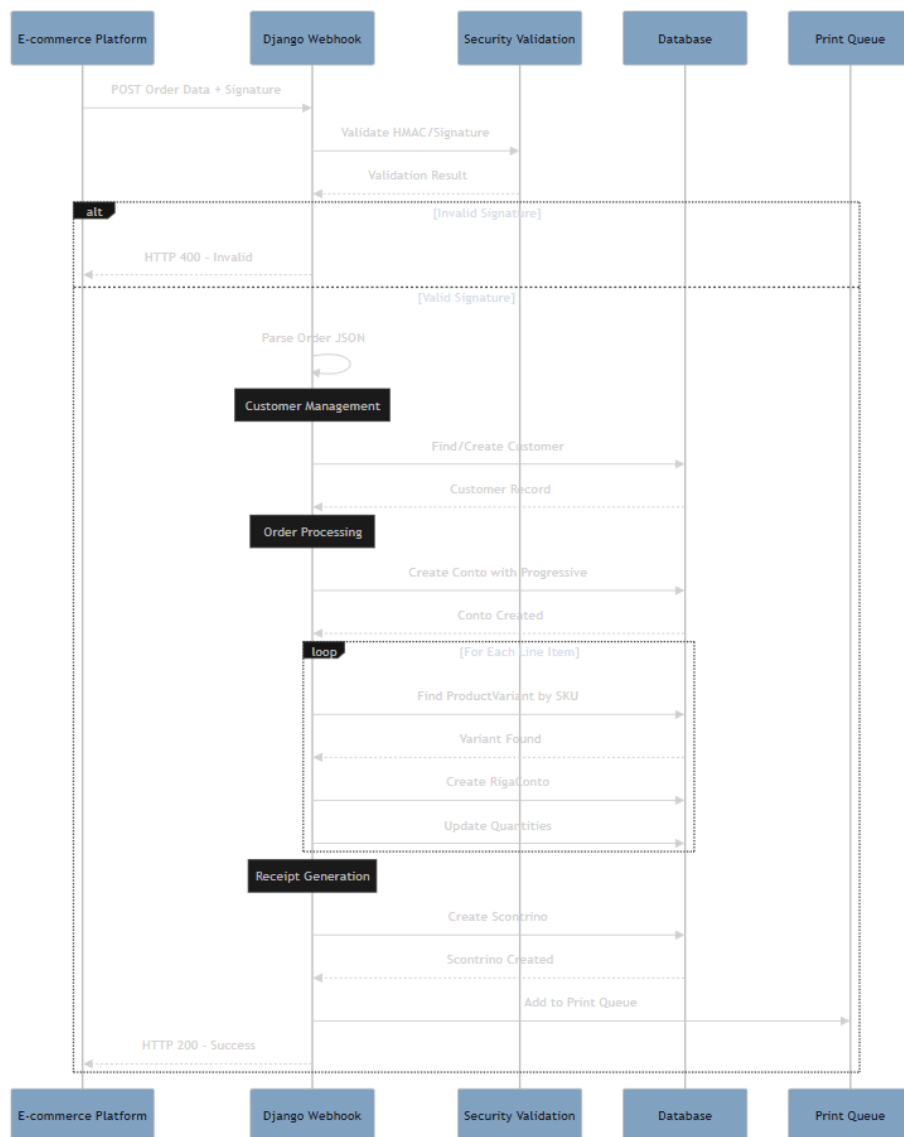


Figura 29: diagramma di sequenza per i webhooks

5

Cap. 5

Struttura del front-end

Il front-end del sistema gestionale rappresenta l'interfaccia principale attraverso cui gli operatori del negozio, i magazzinieri e gli amministratori interagiscono con le funzionalità del back-end. È stato progettato con l'obiettivo di offrire un'esperienza utente semplice, veloce e intuitiva, anche per utenti non tecnici, senza rinunciare alla completezza e alla reattività delle operazioni.

Sviluppato come Single Page Application (SPA) in React, il front-end è organizzato secondo una struttura modulare e scalabile, con componenti riutilizzabili, gestione dello stato condivisa e supporto alla navigazione dinamica. La grafica è stata pensata per adattarsi al contesto operativo: colori neutri, interfacce chiare e funzioni facilmente accessibili, anche su monitor di cassa e postazioni condivise.

Nel corso di questo capitolo verranno descritti l'organizzazione interna del sito, l'impostazione grafica, le principali pagine funzionali, i plugin utilizzati per la gestione dei contenuti visivi, le modalità di comunicazione con le API back-end, e infine gli aspetti legati alla sicurezza e all'autenticazione degli utenti.

5.1 - Organigramma del sito

L'architettura modulare si articola organizzata attorno a quattro macro-aree funzionali, ciascuna dedicata a specifici aspetti della gestione del negozio. L'autenticazione basata su ruoli prevede funzionalità avanzate riservate agli amministratori e interfacce semplificate per operatori standard.

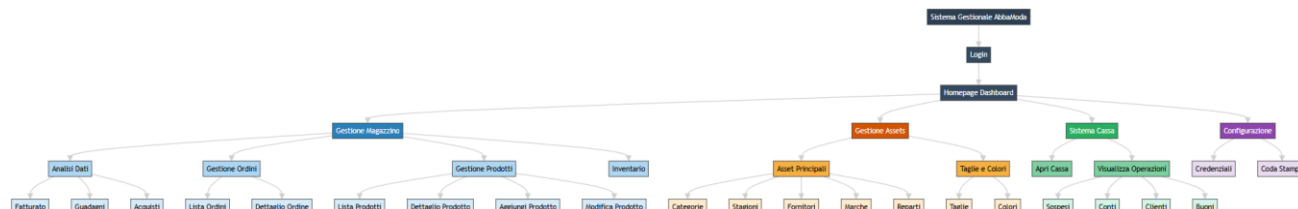


Figura 30: organigramma del sito

5.1.1. - Homepage Dashboard

Punto di accesso principale che fornisce una panoramica generale del sistema con widget informativi e accesso rapido alle funzioni più utilizzate. Presenta statistiche di vendita, alert di sistema e shortcuts per operazioni frequenti.

5.1.2 - Gestione Magazzino

Analisi Dati: raggruppa tre moduli di reportistica: Fatturato per l'analisi delle vendite per periodo, canale e categoria; Guadagni per il calcolo dei margini e profittabilità; Acquisti per il monitoraggio degli approvvigionamenti e costi. Ogni modulo offre filtri avanzati, esportazione dati e grafici interattivi.

Gestione Ordini: include la lista completa degli ordini con filtri per stato, canale e periodo, e la pagina di dettaglio ordine che rappresenta una delle funzionalità più complesse del sistema. La vista dettaglio permette di visualizzare informazioni complete dell'ordine, gestire il carico dei prodotti attraverso scanner barcode o ricerca manuale, modificare quantità e prezzi, aggiungere nuovi articoli direttamente dalla pagina, e gestire stati di evasione e spedizione.

Gestione Prodotti: costituisce il cuore operativo del sistema. La lista prodotti offre ricerca avanzata, filtri multipli e operazioni batch. La funzionalità Aggiungi/Modifica Prodotto rappresenta la pagina più complessa dell'intero sistema, gestendo informazioni base del prodotto, creazione e modifica di varianti multiple per taglia e colore, gestione completa delle immagini locali con possibilità di upload, riordinamento per importanza, impostazione copertina e sincronizzazione automatica con le piattaforme web. Include inoltre gestione delle informazioni specifiche per ogni canale online, mapping delle categorie per marketplace diversi, configurazione prezzi e sconti differenziati.

Inventario: (riservato agli amministratori) fornisce controllo completo delle giacenze con possibilità di correzioni manuali, movimentazioni e riconciliazioni.

5.1.3 - Gestione Assets

Asset Principali: raggruppa la gestione di Categorie, Stagioni, Fornitori, Marche e Reparti. Ogni modulo segue un pattern CRUD standard con validazioni specifiche e relazioni tra entità. Le categorie supportano strutture gerarchiche, mentre fornitori e marche includono informazioni di contatto complete.

Taglie e Colori gestisce le varianti prodotto con supporto per sistemi di taglie personalizzati, equivalenze tra standard diversi, e palette colori con codici hex per uniformità visiva.

5.1.4 - Sistema Cassa

Apri Cassa: rappresenta l'interfaccia POS più sofisticata del sistema. Supporta lettura barcode tramite pistola scanner per caricamento automatico prodotti, ricerca manuale con autocompletamento, modifica quantità e prezzi in tempo reale, applicazione sconti per singolo articolo o totale, gestione clienti con ricerca rapida e creazione, selezione metodi di pagamento multipli, gestione buoni sconto e carte regalo, sospensione vendite per completamento successivo, e stampa automatica scontrini con integrazione alla coda stampa.

Visualizza Operazioni: include Sospesi per recuperare vendite non finalizzate, Conti per storico transazioni, Clienti per anagrafica completa e Buoni per gestione carte regalo e buoni sconto con controllo validità e utilizzo.

5.1.5 - Configurazione

Credenziali: gestisce l'integrazione con piattaforme esterne attraverso configurazione API key, token OAuth, e parametri di connessione per Shopify, eBay e altri marketplace, con test di connettività e rinnovo automatico credenziali.

Coda Stampa: monitora lo stato delle stampe con visualizzazione real-time, possibilità di ristampa, annullamento operazioni e gestione errori, supportando sia etichette prodotto che scontrini fiscali.

5.2 - Impostazione e tema grafico

Il sistema frontend è costruito su un'architettura grafica moderna che combina Bootstrap come framework CSS principale con Reactstrap per i componenti React integrati. Per le icone, il sistema integra Remix Icons e FontAwesome, garantendo coerenza visiva e un set completo di simboli. Le tabelle interattive sono gestite da TanStack React Table, mentre i grafici utilizzano Chart.js con React-ChartJS-2 per visualizzazioni dati avanzate. L'interfaccia include componenti specializzati come React-File-Pond per upload file drag-and-drop, React-Select per dropdown avanzati, Flatpickr per selezione date, e React-Toastify per notifiche interne al sito. Per la gestione delle immagini, il sistema integra React-Image-Editor e Fabric.js per editing avanzato. Il tema adotta un design Material-Inspired con palette colori coerente: blu primario per azioni principali, verde per conferme, arancione per avvisi e viola per configurazioni. Gli elementi utilizzano border-radius per un aspetto moderno e transizioni fluide per feedback visivo immediato. L'interfaccia implementa hover effects sofisticati e box-shadow dinamiche

per migliorare l'interattività. I bottoni includono animazioni di pressione e gli elementi interattivi forniscono feedback visivo costante attraverso cambi di colore e dimensione

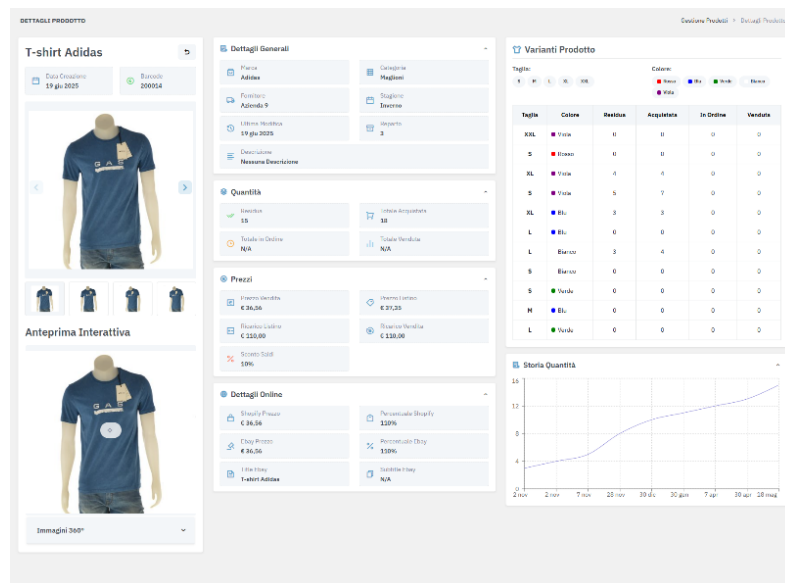


Figura 31: pagina dettaglio di un prodotto

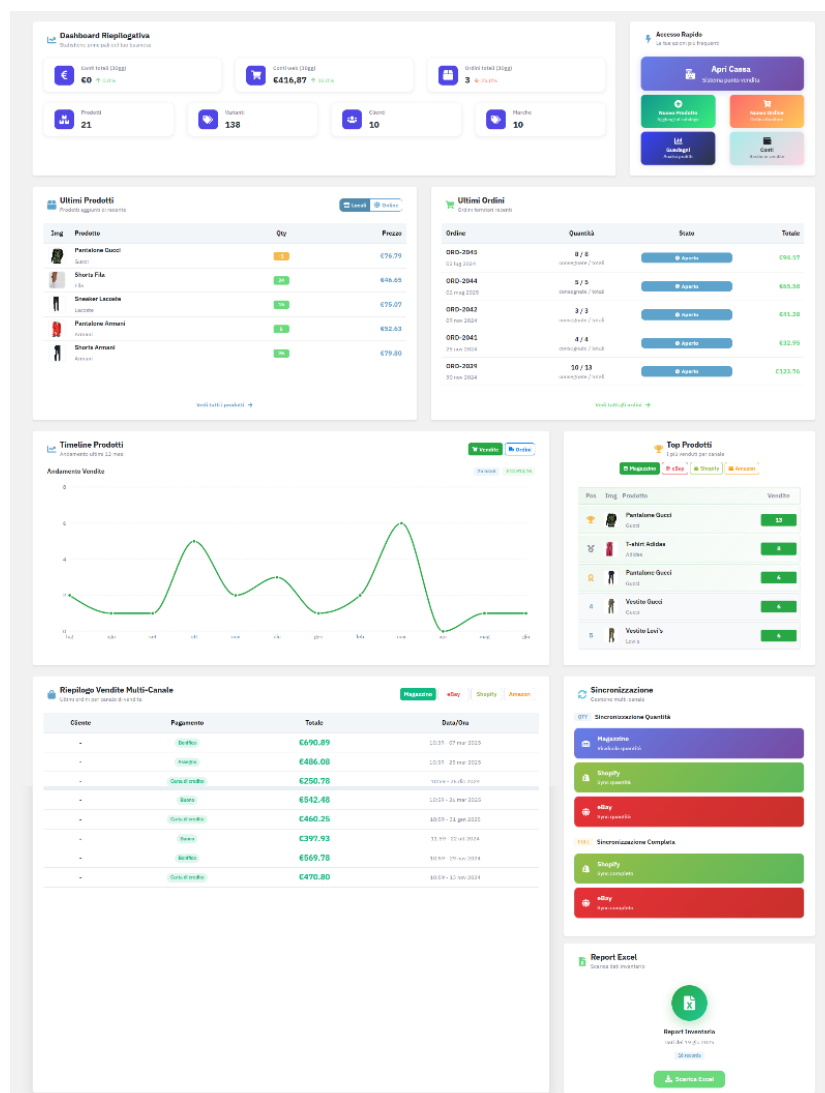


Figura 32: Homepage

5.3 - Interfacciamento con le API del back-end

La comunicazione tra front-end e back-end avviene attraverso con il backend attraverso una serie di servizi specializzati che gestiscono diversi aspetti dell'applicazione. L'architettura è basata su Axios per le chiamate HTTP e implementa un sistema di autenticazione unificato con token management automatico. Il servizio di autenticazione coordina login, logout e gestione dei token attraverso localStorage, implementando controlli automatici di validità e redirect per sessioni scadute. Ogni servizio verifica l'autenticazione prima di effettuare chiamate API e include automaticamente gli header di autorizzazione necessari. La configurazione degli endpoint utilizza variabili d'ambiente per facilitare deployment in ambienti diversi.

Il servizio API principale gestisce tutte le operazioni CRUD del sistema warehouse attraverso funzioni generiche per fetch, create, update e delete. Include oltre 70 endpoint specifici per prodotti, ordini, varianti, clienti, fornitori e tutte le entità del sistema. Implementa gestione automatica degli errori, parametri di paginazione e supporto per upload multipart per le immagini.

I servizi web specializzati per le piattaforme esterne coordinano le integrazioni con marketplace e sistemi di terze parti. Il servizio Shopify gestisce pubblicazione prodotti, sincronizzazione categorie, gestione credenziali API e recupero link prodotti. Il servizio eBay implementa funzionalità analoghe con gestione delle categorie specifiche, aspetti prodotto e credenziali multiple per ambienti diversi.

Il servizio immagini coordina upload, modifica, riordinamento e sincronizzazione delle immagini prodotto. Supporta operazioni drag-and-drop, crop automatico, conversione formati e gestione delle copertine. Include funzionalità per spostare immagini tra locale e web, duplicazione e eliminazione batch.

Il servizio stampa gestisce la coda di stampa per etichette e scontrini attraverso endpoint dedicati per monitoraggio stato, ristampa, annullamento e gestione errori. Implementa polling automatico per aggiornamenti real-time e notifiche di completamento operazioni.

Il servizio contabilità coordina la gestione finanziaria con endpoint per fatturato, guadagni, analisi acquisti e reportistica avanzata. Include filtri temporali, raggruppamenti per categoria e esportazione dati in diversi formati.

Il servizio home gestisce la dashboard principale con widget informativi, statistiche rapide, notifiche sistema e shortcuts per operazioni frequenti. Coordina il caricamento asincrono di dati da diverse fonti per popolare la homepage senza impattare le performance.

6

Cap. 6

Deployment e mantenimento

Lo sviluppo di un sistema gestionale non si è concluso con la scrittura del codice: per essere realmente efficace, deve essere distribuito, aggiornabile e mantenibile nel tempo. Questo capitolo descrive le tecniche che ho adottato per garantire il deployment stabile del sistema nella rete interna del negozio, la gestione delle versioni del codice, l'automazione dei test, e la configurazione di un'infrastruttura scalabile e riutilizzabile tramite Docker.

L'obiettivo è assicurare un flusso di lavoro continuo, in cui ogni modifica possa essere tracciata, testata e rilasciata con il minimo impatto sulle attività quotidiane dell'azienda. Attraverso l'uso di strumenti come Git, Docker, NGINX e pipeline CI/CD, è stato possibile strutturare una soluzione che coniuga semplicità di gestione e solidità architetturale, riducendo il rischio di errori manuali e migliorando l'affidabilità generale del sistema.

6.1 - Versionamento del progetto tramite GIT

Durante lo sviluppo del sistema gestionale, è stato utilizzato Git come sistema di versionamento per organizzare il lavoro in modo chiaro, mantenere traccia delle modifiche nel tempo e poter tornare facilmente a una versione precedente del progetto in caso di errori o regressioni.

Nonostante il progetto sia stato sviluppato individualmente, ho adottato una struttura simile a quella usata nei team di sviluppo, per favorire ordine, scalabilità futura e mantenibilità del codice.

6.1.1 - Organizzazione delle branch

Per gestire le diverse fasi di sviluppo, ho impostato una divisione semplice tra i rami del progetto:

- **main:** contiene la versione stabile e pronta per l'uso del sistema, quella che viene effettivamente distribuita e installata.
- **develop:** include tutte le modifiche in corso, dove vengono testate e integrate nuove funzionalità prima di un rilascio.
- **feature/*:** per ogni nuova funzionalità, ho creato un ramo separato, ad esempio `feature/cassa-resi` o `feature/sync-shopify`, che poi veniva unito a `develop` una volta completato.

In alcuni casi, per correzioni rapide, ho utilizzato anche dei rami `hotfix/*` o `fix/*`, dedicati alla risoluzione di piccoli problemi individuati durante l'uso quotidiano del gestionale.

6.1.2 - Flusso di lavoro personale

Anche lavorando da solo, ho seguito un piccolo schema organizzativo:

1. Apertura di una nuova branch da `develop` per ogni nuova funzionalità.
2. Scrittura del codice con commit frequenti e descrittivi.
3. Una volta terminata la modifica, unione della branch in `develop`.
4. Periodicamente, rilascio su `main` e creazione di un tag (es. `v1.0`) per segnare una versione stabile del sistema.

Ho mantenuto uno stile uniforme nei messaggi di commit per facilitare la lettura dello storico.

6.1.3 - Vantaggi riscontrati

L'uso di Git, anche in un contesto individuale, si è rivelato fondamentale per:

- Mantenere ordine nel progetto anche su sviluppi lunghi o complessi.
- Riprendere funzionalità in sospeso grazie alla gestione separata delle branch.
- Effettuare test senza intaccare la versione stabile.
- Documentare il processo di sviluppo attraverso i commit e i tag.

6.2 - Test automatici delle api del back

Per garantire l'affidabilità del sistema gestionale e ridurre il rischio di errori durante l'evoluzione del progetto, ho implementato diversi livelli di test automatici, in particolare per le API back-end sviluppate con Django e Django REST Framework.

L'obiettivo era verificare in maniera sistematica il corretto funzionamento delle funzionalità principali, validare la risposta del sistema in condizioni reali e simulare l'interazione degli utenti con l'interfaccia front-end.

6.2.1 - Test unitari

I test unitari rappresentano il livello più basilare, ma anche il più importante, dell'architettura di test. Ho utilizzato questi test per verificare il comportamento di singole funzioni o metodi, in particolare quelli legati alla gestione delle quantità (`quantity_services`), alla logica degli ordini e al calcolo del fatturato.

Ogni funzione è stata testata isolatamente, con input controllati e attesi, per garantire il rispetto della logica anche in presenza di casi limite (es. consegne parziali, resi multipli, prodotti senza varianti).

I test sono stati scritti con il framework nativo di Django (TestCase), con una copertura significativa dei moduli.

6.2.2 - Test integrativi

Ho utilizzato questi test per verificare il corretto comportamento delle API, simulando le chiamate reali che il front-end eseguirebbe. Questi test validano non solo le risposte HTTP (200, 400, 403, ecc.), ma anche il contenuto restituito, le operazioni effettuate sul database e la coerenza dei dati modificati.

Ho testato scenari come:

- Creazione di un ordine a fornitore e relative consegne.
- Esecuzione di una vendita tramite API cassa.
- Generazione e riscossione di un buono.
- Pubblicazione di un prodotto su Shopify simulando le risposte API.

Questi test permettono di rilevare problemi nelle interazioni tra i diversi componenti dell'applicazione e assicurano che il sistema risponda correttamente in situazioni reali.

6.2.3 - Test cross-browser (end-to-end)

Per verificare che l'interazione tra back-end e front-end funzionasse correttamente anche a livello utente, ho realizzato test end-to-end utilizzando strumenti come Playwright o Selenium, eseguibili su diversi browser (Chrome, Firefox, Edge).

Questi test simulano azioni complete come:

- Login e autenticazione.
- Apertura della cassa e creazione di un conto.
- Inserimento di prodotti tramite barcode.
- Navigazione tra le pagine di magazzino, inventario e pubblicazione online.

L'obiettivo non era solo testare l'interfaccia, ma validare il flusso di comunicazione tra front-end e back-end, compresi gli aggiornamenti in tempo reale e i token di autenticazione.

6.2.4 - Test di performance

Infine, ho effettuato test di carico e prestazioni sulle API più utilizzate. I test hanno simulato numerose richieste concorrenti per:

- Accesso alla dashboard dei conti.
- Sincronizzazione delle quantità con i canali online.
- Operazioni massicce di pubblicazione di prodotti.

Questi test hanno permesso di individuare i colli di bottiglia e ottimizzare alcune parti lente, migliorando l'esperienza utente in ambienti reali e riducendo il rischio di blocchi.

6.3 - Messa in produzione del sistema tramite Docker

L'adozione di Docker per la distribuzione del sistema gestionale garantisce coerenza tra ambienti di sviluppo e produzione, eliminando problemi di compatibilità e ottimizzando il deployment e manutenzione. Docker crea container isolati che includono codice, dipendenze e configurazioni, assicurando portabilità e scalabilità del sistema.

Ho utilizzato un'architettura multi-container orchestrata tramite Docker Compose, che coordina l'intero stack attraverso un singolo file di configurazione. Questo approccio permette di gestire servizi indipendenti che comunicano attraverso una rete dedicata, facilitando aggiornamenti selettivi e manutenzione modulare.

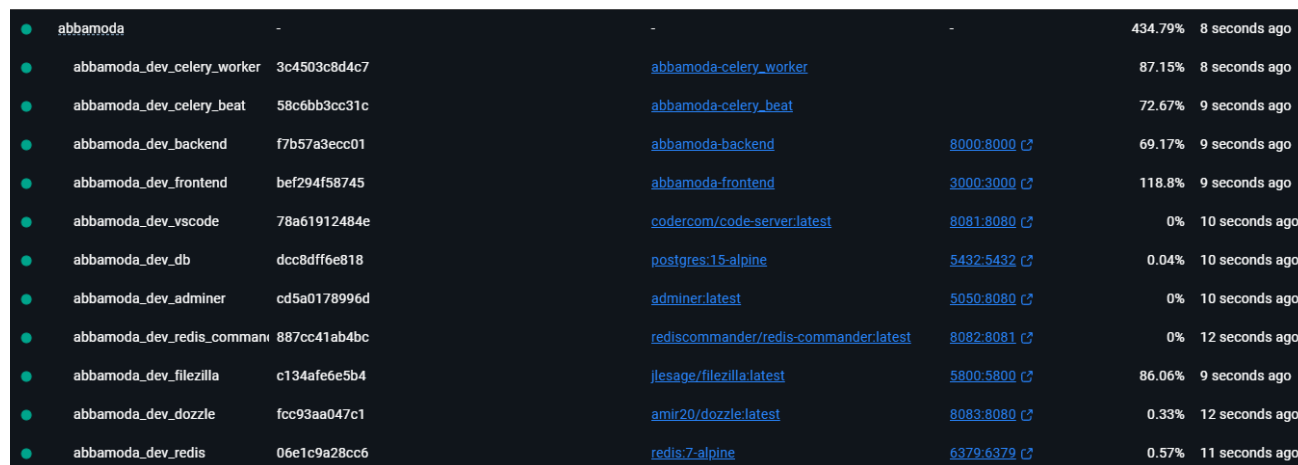
Il sistema include container per database PostgreSQL con storage persistente, cache Redis per sessioni e task queue, back-end Django con Gunicorn per la produzione, worker Celery per task asincrone, scheduler Celery Beat per operazioni periodiche, e front-end React servito tramite development server ottimizzato.

La configurazione implementa volumi persistenti per garantire continuità dei dati: `postgres_data` conserva il database anche durante aggiornamenti container, `redis_data` mantiene cache e sessioni attive, `media_data` preserva immagini e upload utente, mentre `vscode_data` conserva configurazioni dell'ambiente di sviluppo.

Il file `.env` centralizza tutte le variabili di configurazione, includendo porte di servizio, credenziali database, endpoint API, chiavi di sicurezza e parametri di connessione. Questa centralizzazione facilita configurazioni per ambienti diversi e migliora la sicurezza separando credenziali dal codice. La procedura di deployment si riduce a pochi comandi: configurazione variabili ambiente, build automatica delle immagini tramite `docker-compose build`, e avvio dello stack completo. Ho incluso nel sistema controlli di dipendenza che garantiscono l'ordine corretto di avvio dei servizi.

6.4 - Creazione dello stack Docker per servire il sistema gestionale sulla rete interna

I servizi principali comprendono PostgreSQL 15 Alpine per il database con volume persistente, Redis 7 Alpine per caching e message broker, back-end Django con hot-reload per sviluppo, worker e scheduler Celery per task asincrone, e front-end React con polling abilitato per aggiornamenti real-time durante lo sviluppo.



abbamoda	-	-	-	434.79%	8 seconds ago
abbamoda_dev_celery_worker	3c4503c8d4c7	abbamoda-celery_worker		87.15%	8 seconds ago
abbamoda_dev_celery_beat	58c6bb3cc31c	abbamoda-celery_beat		72.67%	9 seconds ago
abbamoda_dev_backend	f7b57a3ecc01	abbamoda-backend	8000:8000 ↗	69.17%	9 seconds ago
abbamoda_dev_frontend	bef294f58745	abbamoda-frontend	3000:3000 ↗	118.8%	9 seconds ago
abbamoda_dev_vscode	78a61912484e	codercom/code-server:latest	8081:8080 ↗	0%	10 seconds ago
abbamoda_dev_db	dcc8dff6e818	postgres:15-alpine	5432:5432 ↗	0.04%	10 seconds ago
abbamoda_dev_adminer	cd5a0178996d	adminer:latest	5050:8080 ↗	0%	10 seconds ago
abbamoda_dev_redis_commander	887cc41ab4bc	rediscommander/redis-commander:latest	8082:8081 ↗	0%	12 seconds ago
abbamoda_dev_filezilla	c134afe6e5b4	jlesage/filezilla:latest	5800:5800 ↗	86.06%	9 seconds ago
abbamoda_dev_dozzle	fcc93aa047c1	amir20/dozzle:latest	8083:8080 ↗	0.33%	12 seconds ago
abbamoda_dev_redis	06e1c9a28cc6	redis:7-alpine	6379:6379 ↗	0.57%	11 seconds ago

Figura 33: configurazione docker

6.4.1 - Configurazione HTTPS con certificati gratuiti

Un requisito essenziale per ricevere webhook da Shopify ed eBay è l'utilizzo di connessioni HTTPS valide. Ho implementato i certificati SSL/TLS gratuiti tramite Let's Encrypt, configurati attraverso un reverse proxy NGINX che gestisce automaticamente generazione e rinnovo certificati. Per ambienti di sviluppo locale, vengono utilizzati certificati self-signed accettati dai dispositivi della rete interna, garantendo crittografia completa delle comunicazioni anche senza dominio pubblico.

6.4.2 - Servizi ausiliari per amministrazione

Adminer offre un'interfaccia web leggera per gestione database con tema Pepa-Linha, permettendo query dirette, backup e amministrazione tabelle senza client esterni. VSCode Server trasforma il browser in un IDE completo, consentendo sviluppo remoto con accesso a tutti i file di progetto, terminale integrato e estensioni, particolarmente utile per modifiche rapide da dispositivi diversi. Redis Commander fornisce monitoraggio real-time della cache e delle code Celery, visualizzando chiavi attive, memoria utilizzata e statistiche performance. Dozzle aggrega logs di tutti i container in un'interfaccia unificata con ricerca full-text, filtri per servizio e streaming real-time, eliminando la necessità di accesso SSH per debugging. Filezilla per poter gestire i file presenti nel progetto.

6.4.3 - Gestione volumi e persistenza

La strategia di persistenza combina volumi per dati critici e bind mount per lo sviluppo. I volumi `postgres_data` e `redis_data` garantiscono continuità database e cache durante aggiornamenti, mentre `media_data` preserva immagini e upload utente. Il volume `vscode_data` mantiene configurazioni IDE e estensioni tra restart. I bind mount collegano directory host ai container: codice sorgente per hot-reload, logs per accesso diretto dall'host, e socket Docker per monitoraggio container. Questa configurazione bilancia prestazioni, persistenza e accessibilità durante sviluppo.

6.4.5 - Rete e sicurezza

La rete `abbamoda_network` isola tutti i servizi in un ambiente controllato, permettendo comunicazione interna tramite nomi container mentre espone solo le porte necessarie all'host. Le porte sono centralizzate nel file `.env` per evitare conflitti e facilitare configurazioni multiple. Il file `.env` centralizza credenziali, endpoint API, configurazioni database e parametri di sicurezza, separando informazioni sensibili dal codice e permettendo configurazioni ambiente-specifiche. Questa centralizzazione semplifica deployment e migliora sicurezza attraverso gestione unificata delle variabili. L'architettura supporta configurazioni ambiente-specifiche che adattano comportamenti per sviluppo o produzione, con entry-point script che gestiscono inizializzazione database, migrazioni automatiche e setup configurazioni, garantendo avvio corretto indipendentemente dall'ambiente target.

6.5 - CI/CD per automatizzare i test e la messa in produzione

Per ridurre i tempi di rilascio, minimizzare il rischio di errori manuali e mantenere un flusso di lavoro efficiente, è stata implementata una pipeline CI/CD (Continuous Integration / Continuous Deployment). Anche in un progetto sviluppato individualmente, automatizzare i passaggi tra sviluppo, test e messa in produzione permette di lavorare in modo più ordinato e professionale.

L'obiettivo era creare un sistema che, ogni volta che viene effettuato un aggiornamento al codice (soprattutto sul ramo `main` o `develop`), esegua automaticamente:

- i test automatici (unitari, integrativi, ecc.),
- la build delle immagini Docker,
- il rilascio controllato del sistema, con restart selettivo dei container aggiornati.

Tutto questo consente di rendere la pubblicazione di nuove funzionalità rapida, sicura e ripetibile.

6.5.1 - Strumenti utilizzati

Per implementare la pipeline ho utilizzato strumenti moderni e gratuiti:

- GitHub Actions: per gestire l'automazione di test e build in risposta a eventi del repository (es. push su `main` o apertura di una pull request).
- Watchtower (locale): tool leggero per aggiornare automaticamente i container nel server di produzione quando una nuova immagine viene caricata.

Flusso di lavoro della pipeline

1. Push sul ramo main o develop: ogni volta che una modifica viene pubblicata su uno dei rami principali, viene attivato automaticamente il flusso CI/CD.
2. Installazione delle dipendenze e setup ambiente: il sistema installa le dipendenze Python e JavaScript necessarie, configura l'ambiente virtuale ed esporta le variabili di test.
3. Esecuzione dei test:
 - I test unitari e integrativi vengono eseguiti in un ambiente pulito.
 - Se anche un solo test fallisce, l'intero processo si interrompe (fail-fast).
4. Build delle immagini Docker: se i test hanno successo, viene eseguita la build delle immagini Docker aggiornate (back-end, front-end, ecc.) usando le configurazioni di produzione.
5. Aggiornamento sul server di produzione:
 - Viene effettuato uno script `docker-compose pull && docker-compose up -d`.

6.5.2 - Vantaggi per la gestione futura

Anche in un progetto gestito da un singolo sviluppatore, l'adozione di un processo CI/CD porta benefici tangibili:

- Affidabilità dei rilasci: ogni aggiornamento è verificato automaticamente prima di essere pubblicato.
- Maggiore velocità di sviluppo: si riducono i tempi tra scrittura del codice e sua pubblicazione.
- Manutenzione semplificata: eventuali criticità possono essere isolate e risolte più facilmente.
- Scalabilità futura: se il progetto venisse ampliato o gestito da più persone, la pipeline CI/CD sarebbe già pronta a supportarlo.

Con questa infrastruttura, il progetto si presenta come una soluzione robusta e professionale, non solo dal punto di vista funzionale, ma anche in termini di mantenimento e aggiornabilità nel tempo.

7

Cap. 7

Conclusione

Lo sviluppo del sistema gestionale descritto ha rappresentato un percorso completo, dall'analisi delle esigenze di una PMI alla realizzazione e distribuzione di una piattaforma moderna, efficiente e integrata. Attraverso l'uso di tecnologie attuali, un'architettura modulare e processi automatizzati, è stato possibile costruire un'applicazione che non solo risponde alle esigenze attuali del negozio, ma è anche pronta a evolversi.

In questo capitolo conclusivo vengono presentati alcuni dati riassuntivi sul sistema realizzato, con particolare attenzione alle sue dimensioni effettive e alle performance ottenute durante l'utilizzo quotidiano. Queste informazioni aiutano a comprendere la scalabilità, la sostenibilità e il reale impatto tecnico del progetto.

7.1 - Numeri e dimensioni del sistema

Il sistema gestionale nel corso di questo progetto ha raggiunto, al termine della sua prima implementazione, una dimensione e una complessità considerevoli. Di seguito vengono riportati i principali dati quantitativi che descrivono lo stato attuale del sistema.

7.1.1 - *Dati gestionali*

Il database principale ospita oltre 50.000 prodotti unici, ciascuno dei quali può avere una o più varianti. Complessivamente, il numero di varianti gestite è di circa 140.000 unità, tutte sincronizzate con le giacenze di magazzino, i canali di vendita e gli ordini in corso.

Nel sistema risultano registrati:

- Più di 1.500 ordini a fornitore, con una media di 5–10 righe per ordine.
- Oltre 20.000 conti di vendita generati nel tempo, ciascuno con i relativi prodotti, sconti, resi e metodi di pagamento.
- Circa 2.300 buoni attivi (tra buoni regalo, resi convertiti e sospesi).
- Una media di 400 prodotti movimentati al giorno, considerando vendite, carichi e resi.

La dimensione complessiva del database attualmente si attesta intorno ai 1,2 GB, considerando dati relazionali, log di operazioni e cronologia delle sincronizzazioni.

7.1.2 - *Struttura software*

Dal punto di vista architetturale, il sistema si compone di:

- 1 front-end React (SPA) suddiviso in circa 40 schermate principali e oltre 100 componenti riutilizzabili.
- 4 API Django: warehouse, webapi, stampanti, img360, per un totale di oltre 180 endpoint REST documentati.
- Una media di 1.500 chiamate API giornaliere, con picchi fino a 4.000 nei giorni di maggior carico (soprattutto online).

7.1.3 - *Media e contenuti web*

Il sistema gestisce in totale circa 180.000 immagini prodotto, suddivise tra:

- immagini originali ad alta risoluzione,
- immagini ottimizzate per il web,
- miniature per la navigazione,
- immagini 360° per la visualizzazione interattiva.

Le immagini sono conservate in una struttura ordinata su disco, per una dimensione complessiva di circa 56 GB, gestite e servite da un container dedicato all'interno della rete interna.

7.1.4 - Sincronizzazione e automazioni

Il sistema esegue ogni settimana una serie di task automatici di sincronizzazione, tra cui:

- aggiornamento delle quantità sui canali online,
- verifica e aggiornamento dei metadati e delle immagini,
- ripubblicazione selettiva dei prodotti aggiornati.

Ogni ciclo di sincronizzazione coinvolge in media oltre 15.000 prodotti, con tempistiche ottimizzate per rispettare i limiti delle API di Shopify ed eBay.

7.1.5 - Utenti e sicurezza

Il sistema è attualmente utilizzato da 12 account attivi, divisi per reparto (cassa, magazzino, amministrazione, online), con accessi differenziati e autenticazione protetta da token temporanei.

Tutti gli accessi avvengono dalla rete interna del negozio, e con possibilità di backup del database e dei media.

7.2 - Performance del back-end e del front-end

La valutazione delle performance è un aspetto essenziale per garantire che il sistema non solo sia funzionalmente corretto, ma anche in grado di sostenere l'uso quotidiano da parte degli operatori, senza rallentamenti né blocchi, anche in condizioni di carico elevato.

Le seguenti considerazioni si basano su test condotti in ambiente reale e simulato, sia nella rete interna del negozio che in ambiente di staging, con dati realistici in termini di quantità di prodotti, richieste simultanee e interazioni tra i moduli.

7.2.1 - Performance del back-end

Il back-end, sviluppato in Django e strutturato in microservizi containerizzati, ha mostrato ottimi risultati in termini di reattività e stabilità, grazie anche all'adozione di ottimizzazioni come:

- utilizzo di `select_related` e `prefetch_related` per ridurre il numero di query
- caching di chiamate ripetitive
- uso di database indicizzati sulle tabelle più critiche (prodotti, conti, ordini)
- suddivisione dei servizi in 4 API dedicate, per evitare colli di bottiglia

Le metriche registrate durante i vari test sono le seguenti:

- Tempo medio di risposta per le API più utilizzate (cassa, magazzino): 120–180 ms
- Tempo massimo registrato sotto carico simulato (100 utenti concorrenti): < 450 ms
- Percentuale di successo delle richieste API in condizioni normali: 99,9%
- Utilizzo medio della CPU del container Django: 15–18%, con picchi controllati durante operazioni massicce

7.2.2 - Performance del front-end

Il front-end, sviluppato in React e servito da un container NGINX ottimizzato, è stato progettato come Single Page Application (SPA) per ridurre il caricamento delle pagine e migliorare la fluidità dell'interazione. Le varie ottimizzazioni adottate:

- caricamento asincrono dei componenti e delle immagini
- gestione dello stato globale tramite Context e cache locale
- minimizzazione e compressione dei file .js e .css
- prefetch dei dati critici al login per evitare ritardi durante l'uso

Le metriche registrate durante i test sono le seguenti:

- Tempo medio di caricamento iniziale (prima visita): ~1,8 secondi su rete locale
- Tempo di navigazione tra pagine già caricate: < 100 ms (istantaneo)
- Utilizzo medio della memoria lato client: < 200 MB, anche durante operazioni complesse come carichi o vendite

L'esperienza utente è risultata fluida anche su dispositivi di fascia media, grazie alla leggerezza del client e al basso numero di ricaricamenti completi.

7.2.3 - Performance complessiva e affidabilità

Il sistema nel suo complesso ha mostrato un ottimo livello di affidabilità, con un uptime registrato superiore al 99,8% durante le settimane di test. I task automatici di sincronizzazione non hanno mai interferito con l'uso quotidiano, grazie alla loro esecuzione programmata in orari notturni o a bassa attività.

Le prestazioni in rete locale sono risultate superiori rispetto a sistemi SaaS tradizionali, anche grazie all'eliminazione della latenza di rete esterna. Questo ha migliorato notevolmente l'efficienza degli operatori di magazzino e cassa.

In conclusione, le performance rilevate sono pienamente soddisfacenti per le esigenze operative della PMI, e il sistema è pronto anche per una futura espansione in termini di utenti, prodotti e canali di vendita.

Colophon

Questa tesi è stata redatta utilizzando Microsoft Word su sistemi operativi Windows 11 Pro, lavorando da più postazioni fisse, sia in ambito domestico che lavorativo. Il documento è stato impaginato in formato A4, con margini standard e interlinea singola. Il font scelto per il corpo del testo è Aptos, selezionato per la sua chiarezza e leggibilità anche su schermi ad alta risoluzione.

Tutti i diagrammi e le immagini presenti nella tesi sono stati realizzati autonomamente: i diagrammi sono stati sviluppati in Mermaid.js e convertiti in formato immagine, mentre le schermate delle interfacce sono state ottenute direttamente dal front-end realizzato nel progetto. La gestione del codice sorgente è stata organizzata tramite il sistema di versionamento Git, come descritto nel Capitolo 6.1.

La stesura, l'impaginazione, l'analisi tecnica e lo sviluppo software sono stati interamente svolti in autonomia.

Bibliografia

1. Casaleggio Associati, *Report E-commerce Italia 2024*, Milano, 2023.
2. Galattico.ai, *Digitalization of Italian SMEs: current status and European comparison*, Roma, 2024.
3. Mordor Intelligence, *Multichannel Order Management Market Size & Growth, 2025–2030*, 2024.
4. Spherical Insights, *Italy Warehouse Management Software Market Insights Forecasts to 2033*, Feb 2025.
5. Extensiv, *Inventory Visibility: 2025 Trends, Challenges, and How to Conquer Them*, Jan 11, 2024.
6. JLL Research, *Italian Logistics Snapshot | Q4 2023*, 2023.
7. Netcomm – Politecnico di Milano, *E-commerce in Italia: canali di vendita e transazioni digitali*, Osservatorio e-Commerce B2c, 2023.
8. Osservatori Digital Innovation – School of Management (Polimi), *Social Commerce Trends 2023*, Milano, Dicembre 2023.
9. Stack Overflow, *Developer Survey 2024 – Dati sull’adozione di React (47% front-end developers)*.
10. Stack Overflow, *Developer Survey 2024, maggio 2024 (38 % delle vulnerabilità SQL vs 5 % con ORM)*
11. Stack Overflow *Developer Survey 2024 – Statistiche su Django come framework popolare*.
12. Socket.dev, *2023 npm Retrospective on Growth – Statistiche sul riutilizzo del codice React Native (fino all’80%)*.
13. JetBrains, *State of Developer Ecosystem Report 2024*, aprile 2024 (45 % ORM adoption).
14. JetBrains, *State of Developer Ecosystem Report 2024 – Dati sull’adozione di Python e ORM*.
15. JetBrains, *Developer Ecosystem Survey 2023*.
16. Python Software Foundation & JetBrains, *Python Developers Survey 2023*, febbraio 2024 (55 % riduzione tempo migrazioni).
17. Django Documentation & GitHub – *Informazioni sull’integrazione tra Django e React nel tuo progetto*.
18. GitHub – *Metriche sulla popolarità di Django (oltre 75k stelle)*.

19. Python Software Foundation & JetBrains, Python Developers Survey 2023 – Crescita dell’ecosistema Django.
20. Shopify Dev Docs, *Using GraphQL Admin API*, shopify.dev
21. GraphQL Foundation, *GraphQL Specification*, graphql.org
22. Altostra, *GraphQL vs REST API – Use Cases and Performance*, 2023
23. Apollo, *State of GraphQL 2023 Report*.
24. Postman, State of the API Report 2023, postman.com/state-of-api.
25. Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, 2000.
26. Swagger/OpenAPI Specification, swagger.io/specification.