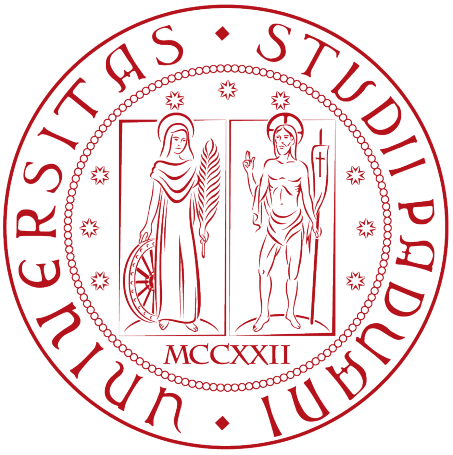
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Un sistema per la visualizzazione di schede prodotto nelle confezioni: sviluppo con approccio Domain-Driven Design

Tesi di Laurea

Relatore

Prof. Paolo Baldan

Laureando

Matteo Cescon, Matricola 2009984

Anno Accademico 2022-2023

Matteo Cescon, Matricola 2009984: *Un sistema per la visualizzazione di schede prodotto nelle confezioni: sviluppo con approccio Domain-Driven Design,* Tesi di Laurea, © Dicembre 2023.

Sommario

Il presente documento descrive il lavoro svolto dal laureando Matteo Cescon durante il periodo di tirocinio presso l'azienda Sogea S.r.l. tra settembre e novembre 2023. L'obiettivo fissato consiste nell'implementazione di una web app commissionata da un cliente operante nel settore delle confezioni. Il sistema si occupa della visualizzazione e della gestione di schede prodotto: queste dovranno rappresentare delle camicie e tutti i relativi dettagli, in modo che, una volta visualizzate nei laboratori, possano esserci tutte le informazioni per la produzione. Il prodotto, basato su un'architettura a microservizi, è stato sviluppato attraverso l'approccio Domain-Driven Design. La web app si basa sul framework Angular e Bootstrap con un'interfaccia responsive. Lo sviluppo del backend, invece, sfrutterà il framework .Net Core e implementerà diversi design pattern di supporto al Domain-Driven Design come ad esempio CQRS, Repository Pattern. Per la persistenza dei dati è stato utilizzato PostgreSQL. I linguaggi principali utilizzati nello sviluppo sono stati TypeScript per il frontend e C# per il backend.

Ringraziamenti

Vorrei esprimere prima di tutto la mia gratitudine al Prof. Paolo Baldan, per i consigli, il sostegno e l'aiuto che mi ha fornito durante la stesura della tesi.

Successivamente, ringrazio particolarmente il tutor aziendale Simone Ronchin per la disponibilità e la pazienza dimostrate durante il periodo di tirocinio. Ringrazio inoltre tutti i colleghi per l'accoglienza in un ambiente ospitale e stimolante; un ringraziamento anche all'azienda Sogea S.r.l. per avermi ospitato.

Desidero ringraziare i miei genitori per la motivazione e l'aiuto fornitomi, ma anche per avermi concesso l'opportunità di svolgere questo percorso di studi, dal momento che senza il loro aiuto non sarebbe stato possibile.

Concludendo, un grazie a tutti gli amici per essermi stati vicini durante questi anni. Ringrazio in particolar modo la mia fidanzata Michelle, per avermi sostenuto durante i periodi più difficili.

Padova, Dicembre 2023 Matteo Cescon, Matricola 2009984

Indice

1 Introduzione **1**

1.1 Il progetto . . . 1
1.2 Soluzione scelta . . . 1
1.3 Strumenti utilizzati . . . 2
1.4 Descrizione del prodotto ottenuto . . . 3
1.5 Organizzazione del documento . . . 4
1.5.1 Struttura . . . 4
1.5.2 Convenzioni tipografiche . . . 4

2 Analisi dei requisiti **5**

2.1 Funzioni principali . . . 5
2.2 Utenti . . . 5
2.3 Casi d'uso . . . 6
2.3.1 UC 1 - UC 3: Gestione utente e lingua . . . 6
2.3.2 UC 4 - UC 10: Gestione dei clienti . . . 8
2.3.3 UC 11 - 17: Gestione delle schede . . . 13
2.3.4 UC 18 - 31: Gestione di note, allegati, immagini e accessori . . . 18
2.3.5 UC 32 - 36: Gestione di attributi generici . . . 21
2.3.6 UCE 1 - 2: Errori riguardanti l'inserimento di testo . . . 23
2.4 Tracciamento requisiti . . . 24
2.4.1 Requisiti funzionali . . . 24
2.4.2 Requisiti qualitativi . . . 25
2.4.3 Requisiti di vincolo . . . 26
2.4.4 Requisiti prestazionali . . . 26

3 Progettazione **27**

3.1 Domain-Driven Design . . . 27
3.2 Schema generale . . . 31
3.3 Frontend . . . 32
3.4 Backend . . . 34
3.4.1 Pattern e metodologie . . . 34
3.4.2 Database . . . 35
3.4.3 API . . . 35
3.4.4 Struttura . . . 35

4 Realizzazione **37**

4.1 Metodo di lavoro . . . 37
4.1.1 Versionamento e code review . . . 37
4.1.2 Boards . . . 38
4.2 Frontend . . . 39

4.2.1 Servizi e chiamate delle API . . . 39
4.2.2 Internationalization . . . 40
4.2.3 Routing . . . 40
4.2.4 Autenticazione . . . 42
4.2.5 Navbar . . . 42
4.2.6 Menù e sottomenù . . . 43
4.2.7 Tabelle e liste . . . 43
4.2.8 Dettaglio . . . 45
4.2.9 Modali . . . 49
4.3 Backend . . . 50
4.3.1 Dominio . . . 50
4.3.2 DTO . . . 51
4.3.3 Command . . . 52
4.3.4 Query . . . 53
4.3.5 ApplicationService . . . 54
4.3.6 Infrastructure . . . 54
4.3.6.1 Persistence . . . 54
4.3.6.2 Query . . . 55
4.3.7 Swagger . . . 57
4.3.8 Jasper . . . 57
4.4 Deploy . . . 58

5 Conclusioni **59**
5.1 Valutazione strumenti utilizzati . . . 59
5.2 Miglioramenti . . . 59
5.3 Valutazione personale . . . 60

Glossario **61**

Bibliografia **63**

Elenco delle figure

1.1 Logo Azure DevOps . . . 2
1.2 Logo Fork . . . 2
1.3 Loghi di Visual Studio (a) e di VS Code (b) . . . 2
1.4 Logo pgAdmin/PostgreSQL . . . 2
1.5 Loghi di Docker Desktop (a) e di Portainer (b) . . . 3
1.6 Logo Jaspersoft Studio . . . 3

2.1 UC 1: Login . . . 6
2.2 UC 5: Visualizzazione lista clienti . . . 8
2.3 UC 6: Visualizzazione dettaglio cliente . . . 9
2.4 UC 6.1: Visualizzazione lista note . . . 10
2.5 UC 6.2: Visualizzazione lista allegati . . . 10
2.6 UC 6.3: Visualizzazione lista immagini . . . 11
2.7 UC 12: Visualizzazione lista schede . . . 13
2.8 UC 13: Visualizzazione dettaglio scheda . . . 14
2.9 UC 13.3: Visualizzazione singolo accesorio . . . 16
2.10 UC 15: Stampa scheda . . . 17

3.1 Aggregato Stagione . . . 27
3.2 Aggregato Cliente . . . 28
3.3 Aggregato LineaProdotto . . . 28
3.4 Aggregato Scheda . . . 28
3.5 Aggregati riguardanti l'accessorio . . . 29
3.6 Aggregati Collezione e TabellaMisure . . . 29
3.7 Aggregato Polso . . . 29
3.8 Aggregato Collo . . . 29
3.9 Aggregato CategoriaMerceologica . . . 30
3.10 Aggregato Stato . . . 30
3.11 Schema generale . . . 31
3.12 Angular . . . 32
3.13 Linguaggi utilizzati da un componente Angular . . . 32
3.14 Librerie del frontend . . . 33
3.15 Strumenti del backend . . . 34
3.16 EFCore . . . 35
3.17 Strutta dei file del backend . . . 36

4.1 Branch visualizzati in Azure DevOps . . . 38
4.2 Chiamata che ottiene i clienti attivi . . . 39
4.3 Servizio per le API riguardanti i clienti . . . 39
4.4 Esempi di un errore (a) e di un toast (b) . . . 40

4.5 Esempi i18n . . . 40
4.6 Utilizzo nell'HTML . . . 40
4.7 Gestione del routing . . . 41
4.8 Definizione percorsi figli e parametrizzazione . . . 41
4.9 Inserimento del parametro nell'URL . . . 41
4.10 Funzione di autenticazione . . . 42
4.11 Pagina del login . . . 42
4.12 home.component.html . . . 42
4.13 Navbar . . . 43
4.14 Navigazione principale . . . 43
4.15 Esempio di una kendo grid . . . 44
4.16 HTML di una kendo grid . . . 44
4.17 Grafica con una singola colonna . . . 45
4.18 Rappresentazione a singola colonna . . . 45
4.19 Funzione scatenata al click su una riga . . . 45
4.20 Tab generale di un collo . . . 46
4.21 Lista delle note . . . 46
4.22 Tab dell'immagine . . . 47
4.23 image-list-item.component.html . . . 47
4.24 Grafica tab immagini . . . 47
4.25 Tab generale della scheda . . . 48
4.26 Grafica tab accessori . . . 48
4.27 Grafica tab polso . . . 48
4.28 Grafica tab collo . . . 49
4.29 Dialog per l'inserimento di uno stato . . . 49
4.30 Esempio di dialog . . . 49
4.31 Esempio di aggregate root . . . 50
4.32 Esempio di codice di un aggregate root . . . 50
4.33 Gestione delgli eventi . . . 51
4.34 Esempio di codice di un value object . . . 51
4.35 Esempio di interfaccia repository . . . 51
4.36 Esempio DTO utilizzato per un payload . . . 51
4.37 Esempio di un filtro . . . 52
4.38 Esempio DTO utilizzato per il body di una risposta . . . 52
4.39 Esempio di un command . . . 52
4.40 Esempio di un command handler . . . 53
4.41 Query generali . . . 53
4.42 Esempio di una query specifica . . . 53
4.43 Esempio di handler . . . 54
4.44 Esempio di API . . . 54
4.45 Esempio di repository . . . 55
4.46 Esempio parziale di configurazione . . . 55
4.47 Esempio del modello di cliente . . . 55
4.48 Esempio dell'utilizzo di Automapper . . . 56
4.49 Esempio della gestione di una query . . . 56
4.50 Query handler per la lista delle schede . . . 56
4.51 Swagger . . . 57
4.52 API per la creazione del report . . . 57

Elenco delle tabelle

2.1 Tabella dei requisiti funzionali 25
2.2 Tabella dei requisiti qualitativi 25
2.3 Tabella dei requisiti di vincolo 26

Capitolo 1

Introduzione

1.1 Il progetto

Il progetto è stato commissionato all'azienda ospitante Sogea S.r.l. da un'impresa operante nel settore delle confezioni, in particolare nella produzione di camice da uomo.
La richiesta è un sistema che gestisca delle schede prodotto rappresentanti le camicie: dovranno definire tutti i dettagli necessari, visto che queste schede saranno consultate dai dipendenti nei vari laboratori di produzione dell'azienda, in modo da sapere quali caratteristiche presenta il capo in questione. Dovranno essere definiti svariati attributi di una camicia, come ad esempio il tessuto, il modello, quali taglie disponibili, che tipo di polso e colletto avranno, eventuali accessori ecc. In particolare, ad ogni scheda, saranno assegnati obbligatoriamente un cliente e una stagione. Anche i clienti dovranno essere degli oggetti complessi personalizzabili.
Gli utenti inoltre devono poter essere in grado di stampare la scheda di produzione e l'applicazione dev'essere utilizzabile anche da dispositivi mobile, quindi sarebbe ottimale anche un'interfaccia che si adatti ad ogni tipo di schermo.
Il lavoro da svolgere riguardava quasi interamente l'implementazione: questo perché le specifiche principali erano già state definite durante degli incontri tra l'azienda ospitante e il cliente prima del periodo di tirocinio. Inoltre, durante lo svolgimento ci sono stati altri incontri per chiarire dubbi e orientare lo sviluppo verso la direzione corretta. Il prodotto presentato in questo documento non è tuttavia il risultato finale, che è ancora in fase di sviluppo, ma è una versione parziale che comunque è già stata consegnata al cliente.

1.2 Soluzione scelta

Seguendo una soluzione utilizzata in passato dall'azienda ospitante si è deciso di sviluppare un'applicazione web, in modo da poter accedere facilmente al prodotto da qualsiasi dispositivo connesso alla rete interna del cliente. Il sistema ha una distinzione netta tra frontend e backend, in modo da poter separare le due parti e facilitare lo sviluppo.

- **Frontend**: la parte del software che interagisce con l'utente; il suo compito è di gestire tutta la parte grafica di un programma e passare al *backend* eventuali richieste dell'utente, in modo che vengano elaborate. È sviluppato tramite il framework Angular, che permette la creazione di webapp reattive.
- **Backend**: la parte del software nascosta all'utente, che si occupa di effettuare le operazioni vere e proprie; è l'addetto a eseguire le richieste effettuate dal *frontend*: ad esempio si occupa delle interazioni con eventuali database o lo svolgimento di elaborazione dati. Questo si basa sul framework .NET Core e utilizza il linguaggio C#.

Per le comunicazioni tra i componenti è stato utilizzato il protocollo [G]HTTP con stile architetturale [G]REST. Si è deciso inoltre l'utilizzo di [G]container, in modo da poter eseguire i vari componenti del sistema su qualsiasi macchina e per facilitare il deploy nella macchina del cliente.

1.3 Strumenti utilizzati

Azure DevOps Azure DevOps è un servizio offerto da Microsoft che presenta appunto svariate funzionalità nell'ambito [G]DevOps. In particolare sono state usate le [G]Board per la pianificazione del lavoro e la gestione delle [G]repository Git.

Figura 1.1: Logo Azure DevOps

Fork Fork è un [G]software che fornisce una interfaccia grafica per gestire l'utilizzo di [G]Git. Grazie a esso è possibile effettuare tutte le più comuni operazioni di un sistema di controllo di versione senza dover utilizzare l'interfaccia a linea di comando.

Figura 1.2: Logo Fork

IDE Visual Studio è un [G]Integrated Development Environment(IDE) multilinguaggio sviluppato da Microsoft. Dispone anche un sistema di versionamento e consente di installare componenti aggiuntivi per ottenere altre funzionalità. È uno degli IDE più utilizzati e più completi in ambito .NET. È stato utilizzato per lo sviluppo del backend, utilizzando il linguaggio C#.
Visual Studio Code è invece uno degli IDE più utilizzati in generale, vista la sua leggerezza e versatilità. In particolare è progettato per lo sviluppo JavaScript e Web. Infatti è stato utilizzato per il frontend, che è scritto in TypeScript.

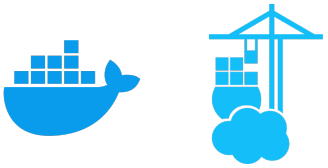
(a) **(b)**

Figura 1.3: Loghi di Visual Studio (a) e di VS Code (b)

pgAdmin pgAdmin è un interfaccia grafica per facilitare l'amministrazione di basi di dati PostgreSQL. Grazie a questo strumento è possibile creare, popolare e interrogare un database.

Figura 1.4: Logo pgAdmin/PostgreSQL

Gestione dei container Docker Desktop è un software per la creazione di container, utilizzato durante lo sviluppo per l'esecuzione della base di dati in locale, in modo da poter testare il backend senza dover effettuare il deploy. Portainer è un software per la gestione di container, immagini, network e volumi. È stato utilizzato principalmente per il deploy dello stack dei container nei diversi ambienti.



(a) (b)

Figura 1.5: Loghi di Docker Desktop (a) e di Portainer (b)

Jaspersoft Studio Jaspersoft Studio è un software per la progettazione di modelli di report. Grazie a questo programma sono stati creati dei modelli di stampa per le schede prodotto.

Figura 1.6: Logo Jaspersoft Studio

1.4 Descrizione del prodotto ottenuto

Il prodotto finale, come detto in precedenza, è ancora in fase di sviluppo. Infatti tra i nuovi bisogni del cliente e gli aggiustamenti per ottenere un sistema più adatto alle richieste, si stanno aggiungendo nuove funzionalità e modifiche da apportare.
Attualmente è possibile gestire tutte le funzionalità principali richieste. Il sistema è un'applicazione accessibile tramite i principali browser che consente tramite alcuni bottoni la navigazione tra una serie di tabelle che permettono di creare, modificare ed eliminare i vari oggetti. La parte più importante è la gestione delle schede vere e proprie: oltre all'elenco di tutti gli oggetti è necessaria una pagina dedicata al dettaglio di una singola scheda, in quanto è un oggetto complesso che non può essere gestito tramite una singola riga. Tramite questa pagina è possibile stampare la scheda (cioè ottenere un file PDF dai dati che la compongono), visualizzarla e gestirne ogni aspetto. Tuttavia non è questa l'unica che ha richiesto un'attenzione particolare: ad esempio anche la gestione dei clienti ha richiesto la definizione di un oggetto complesso.

1.5 Organizzazione del documento

1.5.1 Struttura

Capitolo 2: analizza e descrive i requisiti e i casi d'uso del progetto.

Capitolo 3: descrive gli aspetti tecnici del prodotto software, indicando pattern e tecnologie utilizzate.

Capitolo 4: illustra come il prodotto è stato sviluppato.

Capitolo 5: viene analizzato il prodotto e in particolare il lavoro svolto, approfondendo eventuali miglioramenti possibili.

1.5.2 Convenzioni tipografiche

In questo documento:

- acronimi, abbreviazioni, termini tecnici o di uso generalmente non comune saranno evidenziati e definiti nel glossario, penultima sezione del documento, prima della bibliografia;
- la prima volta che il termine viene nominato nel documento avrà come apice una lettera G, ^{G}esempio.

Capitolo 2

Analisi dei requisiti

2.1 Funzioni principali

La funzione principale del sistema è di gestire una serie di tabelle che contengono i dati necessari per la creazione di schede prodotto. Ogni tabella deve poter compiere in genere le stesse operazioni: inserimento, modifica, eliminazione.
Le tabelle principali gestiscono i clienti e le schede prodotto: per gli elementi di queste tabelle si deve poter accedere a dei dettagli aggiuntivi. Queste hanno in comune una lista di allegati e una di immagini.
In particolare un cliente ha i seguenti attributi oltre a quelli già citati:

- lista di note;
- descrizione;
- priorità;
- linea prodotto.

Invece una scheda prodotto ha i seguenti attributi oltre a quelli già citati:

- modello;
- stato;
- tessuto;
- stagione;
- categoria merceologica;
- tabella delle misure;
- codice articolo cliente;
- area di lavoro;
- note degli adesivi;
- note di taglio;
- particolari di confezione;
- lista di accessori.

Dev'essere possibile inoltre la gestione di tabelle di:

- stagioni;
- collezioni;
- tabelle misure;
- stati;
- polsi;
- colli;
- linee prodotto.

2.2 Utenti

Per ora l'azienda cliente non prevede differenze di utenti, quindi è stato identificato solo un utente semplice. Sarà d'eccezione il caso in cui un utente non è autenticato: gli unici casi trattati sono il login e il cambio della lingua.

2.3 Casi d'uso

Gli [G]use case normali sono indicati con UC X dove X è un numero progressivo univoco, con eventuali sottocasi che saranno definiti come UC X.X e così via. I casi d'uso relativi agli errori sono indicati con UCE X con le indicazioni per la numerazione che sono le stesse.
Per evitare di essere eccessivamente ridondante e per rendere più leggibile il documento, si è deciso di omettere gli use case che non sono strettamente necessari. Avranno particolari attenzioni le tabelle più complesse:

- tabella clienti;
- tabella schede;
- tabella accessori;
- tabella note;
- tabella allegati;
- tabella immagini.

Le altre tabelle avranno invece degli use case che si riferiscono a un oggetto generico, in modo da comprenderle tutte. Inoltre, sempre per gli stessi motivi i casi d'uso minori sono stati omessi in quanto la descrizione dello use case principale è sufficiente a definire anche quelli minori.

2.3.1 UC 1 - UC 3: Gestione utente e lingua

In questi primi use case vengono gestiti il login e il logout dell'utente, oltre alla possibilità di cambiare la lingua dell'applicazione. Il login è necessario per accedere all'applicazione, invece le altre due funzioni sono sempre disponibili durante l'utilizzo.

UC 1: Login

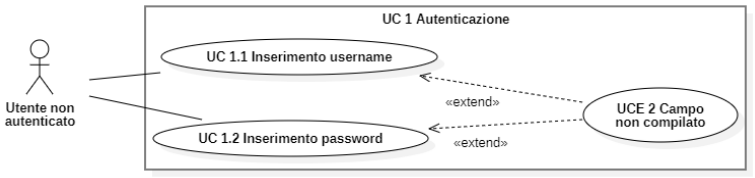


Figura 2.1: UC 1: Login

Attori principali: Utente non autenticato.

Precondizioni: L'utente ha avviato la web app.

Descrizione: L'utente vuole accedere.

Postcondizioni: L'utente ha completato il login.

Scenario principale: L'utente:
1. inserisce il proprio username [UC 1.1];
2. inserisce la propria password [UC 1.2].

Estensioni:
- visualizzazione messaggio d'errore credenziali non valide [UCE 1];

UC 1.1: Inserimento username

Attori principali: Utente non autenticato.

Precondizioni: L'utente ha avviato la web app.

Descrizione: L'utente deve inserire il proprio username per effettuare il login.

Postcondizioni: L'utente ha inserito l'username e può continuare con il login.

Scenario principale:

- l'utente inserisce il proprio username.

Estensioni:

- visualizzazione messaggio d'errore campi non compilati [UCE 2].

UC 1.2: Inserimento password

Attori principali: Utente non autenticato.

Precondizioni: L'utente ha avviato la web app.

Descrizione: L'utente deve inserire la propria password per effettuare il login.

Postcondizioni: L'utente ha inserito la propria password e può procedere con il login.

Scenario principale:

- L'utente inserisce la propria password.

Estensioni:

- visualizzazione messaggio d'errore campi non compilati [UCE 2].

UC 2: Logout

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole effettuare il logout.

Postcondizioni: L'utente non è più autenticato.

Scenario principale:

- l'utente effettua il logout.

UC 3: Cambio lingua

Attori principali: Qualsiasi utente.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole cambiare la lingua.

Postcondizioni: L'utente ha cambiato la lingua.

Scenario principale:

- l'utente cambia la lingua.

2.3.2 UC 4 - UC 10: Gestione dei clienti

In questa sezione verranno trattati tutti gli use case relativi alla gestione dei clienti, in particolare la visualizzazione della lista e le operazioni che possono essere effettuate su un singolo cliente. Sarà anche approfondito il dettaglio di un cliente, con la visualizzazione di tutti i relativi attributi, come ad esempio le liste delle note, degli allegati e delle immagini. La particolarità dei clienti è che possono essere attivati e disattivati, in modo da non eliminarli ma solo nasconderli.

UC 4: Inserimento cliente

Attori principali: Utente autenticato.

Precondizioni: L'utente ha eseguito l'accesso.

Descrizione: L'utente vuole inserire un cliente.

Postcondizioni: L'utente ha inserito un cliente.

Scenario principale:

- l'utente inserisce la descrizione del cliente.

UC 5: Visualizzazione lista clienti

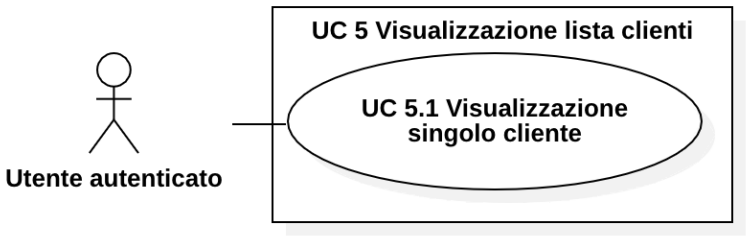


Figura 2.2: UC 5: Visualizzazione lista clienti

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole visualizzare la lista dei clienti.

Postcondizioni: L'utente ha visualizzato la lista dei clienti.

Scenario principale:

- l'utente visualizza la lista dei clienti.

Sottocasi:

- l'utente visualizza un cliente singolo [UC 5.1].

UC 5.1: Visualizzazione cliente singolo

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista dei clienti.

Descrizione: L'utente vuole visualizzare un singolo cliente.

Postcondizioni: L'utente ha visualizzato il singolo cliente.

Scenario principale:

1. l'utente visualizza il logo del cliente;

2. l'utente visualizza la descrizione del cliente;
3. l'utente visualizza la priorità del cliente;
4. l'utente visualizza se il cliente è abilitato o meno.

UC 6: Visualizzazione dettaglio cliente

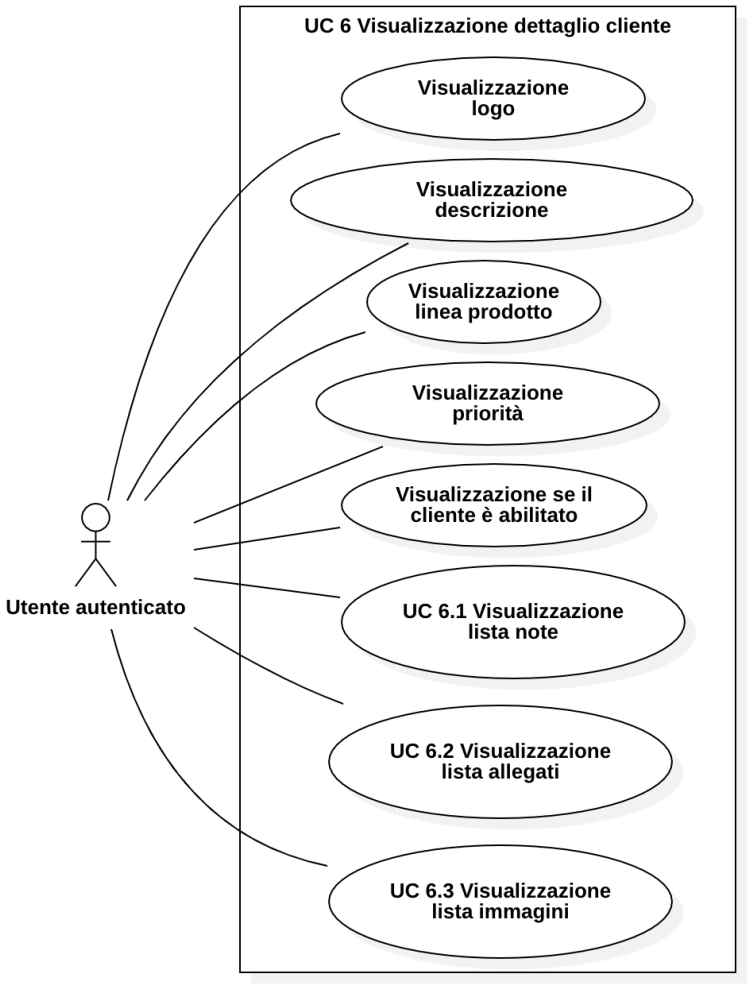


Figura 2.3: UC 6: Visualizzazione dettaglio cliente

Attori principali: Utente autenticato.

Precondizioni: L'utente ha eseguito l'accesso.

Descrizione: L'utente vuole visualizzare il dettaglio di un cliente.

Postcondizioni: L'utente ha visualizzato il dettaglio di un cliente.

Scenario principale:

- l'utente visualizza il logo;
- l'utente visualizza la descrizione;
- l'utente visualizza la priorità;
- l'utente visulizza la linea prodotto;
- l'utente visualizza se il cliente è abilitato o meno;
- l'utente visualizza le note [UC 6.1];
- l'utente visualizza gli allegati [UC 6.2];
- l'utente visualizza le immagini [UC 6.3].

UC 6.1: Visualizzazione lista note

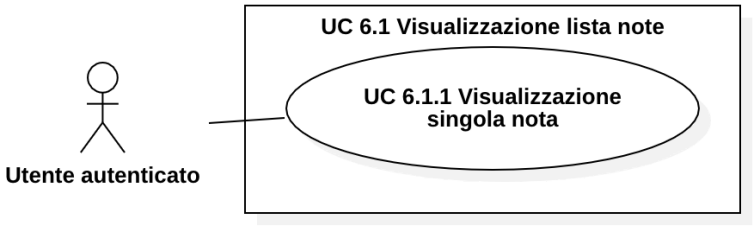


Figura 2.4: UC 6.1: Visualizzazione lista note

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato il dettaglio di un cliente.

Descrizione: L'utente vuole visualizzare la lista delle note di un cliente.

Postcondizioni: L'utente ha visualizzato la lista delle note di un cliente.

Scenario principale:

- l'utente visualizza la lista delle note.

Sottocasi:

- l'utente visualizza una nota singola [UC 6.1.1].

UC 6.1.1: Visualizzazione nota singola

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista delle note di un cliente.

Descrizione: L'utente vuole visualizzare una singola nota di un cliente.

Postcondizioni: L'utente ha visualizzato la singola nota di un cliente.

Scenario principale:

- l'utente visualizza il contenuto della nota.

UC 6.2: Visualizzazione lista allegati

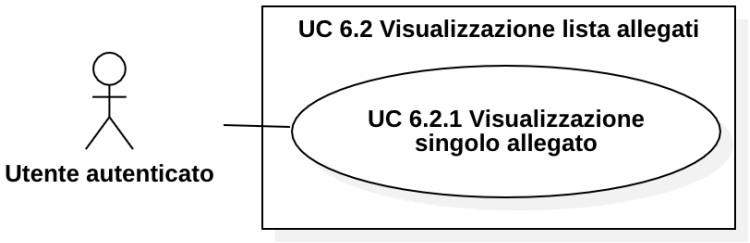


Figura 2.5: UC 6.2: Visualizzazione lista allegati

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato il dettaglio di un cliente.

Descrizione: L'utente vuole visualizzare la lista degli allegati di un cliente.

Postcondizioni: L'utente ha visualizzato la lista degli allegati di un cliente.

Scenario principale:

- l'utente visualizza la lista degli allegati.

Sottocasi:

- l'utente visualizza un allegato singolo [UC 6.2.1].

UC 6.2.1: Visualizzazione allegato singolo

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista degli allegati di un cliente.

Descrizione: L'utente vuole visualizzare un singolo allegato di un cliente.

Postcondizioni: L'utente ha visualizzato il singolo allegato di un cliente.

Scenario principale:

- l'utente visualizza il tipo di allegato;
- l'utente visualizza la descrizione dell'allegato;
- l'utente visualizza il nome del file allegato.

UC 6.3: Visualizzazione lista immagini

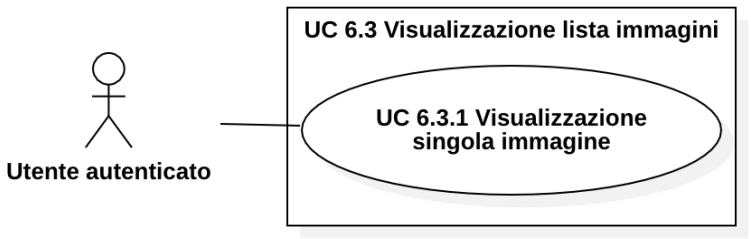


Figura 2.6: UC 6.3: Visualizzazione lista immagini

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato il dettaglio di un cliente.

Descrizione: L'utente vuole visualizzare la lista delle immagini di un cliente.

Postcondizioni: L'utente ha visualizzato la lista delle immagini di un cliente.

Scenario principale:

- l'utente visualizza la lista delle immagini.

Sottocasi:

- l'utente visualizza un'immagine singola [UC 6.3.1].

UC 6.3.1: Visualizzazione immagine singola

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista delle immagini di un cliente.

Descrizione: L'utente vuole visualizzare una singola immagine di un cliente.

Postcondizioni: L'utente ha visualizzato la singola immagine di un cliente.

Scenario principale:

- l'utente visualizza la descrizione dell'immagine;
- l'utente visualizza il tipo dell'immagine.

UC 7: Modifica cliente

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole modificare un cliente.

Postcondizioni: L'utente ha modificato un cliente.

Scenario principale:

1. l'utente modifica la descrizione del cliente;
2. l'utente modifica la priorità del cliente;
3. l'utente modifica il logo del cliente.

UC 8: Attivazione/disattivazione cliente

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole attivare/disattivare un cliente.

Postcondizioni: L'utente ha attivato/disattivato un cliente.

Scenario principale:

- l'utente attiva/disattiva un cliente.

UC 9: Filtraggio clienti

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista dei clienti.

Descrizione: L'utente vuole filtrare la lista dei clienti.

Postcondizioni: L'utente ha filtrato la lista dei clienti.

Sottocasi:

- l'utente seleziona se vuole visualizzare solo i clienti attivi o solo quelli disattivi;
- l'utente inserisce il testo da cercare tra le descrizioni dei clienti.

UC 10: Eliminazione di un cliente

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole eliminare un cliente.

Postcondizioni: L'utente ha eliminato un cliente.

Scenario principale:

- l'utente elimina un cliente.

2.3.3 UC 11 - 17: Gestione delle schede

Le operazioni sono simili a quelle per i clienti ma in questo caso vengono gestite le schede prodotto. La logica è simile a quella degli use case del cliente. Particolarità della scheda prodotto sono i filtri che sono più complessi rispetto a quelli dei clienti. Inoltre, oltre alle liste di allegati e immagini, presenti anche in una scheda, viene trattata la lista degli accessori. Inoltre una scheda deve possedere anche degli oggetti complessi come polso e collo, cioè i dettagli di come devono essere realizzati appunti i polsi e il collo della camicia.

UC 11: Inserimento scheda

Attori principali: Utente autenticato.

Precondizioni: L'utente ha eseguito l'accesso.

Descrizione: L'utente vuole inserire una scheda.

Postcondizioni: L'utente ha inserito una scheda.

Sottocasi:

1. l'utente seleziona il cliente.
2. l'utente seleziona la stagione.
3. l'utente inserisce il modello.
4. l'utente seleziona la categoria merceologica.
5. l'utente inserisce il tessuto.
6. l'utente seleziona la collezione.
7. l'utente inserisce la descrizione.

UC 12: Visualizzazione lista schede

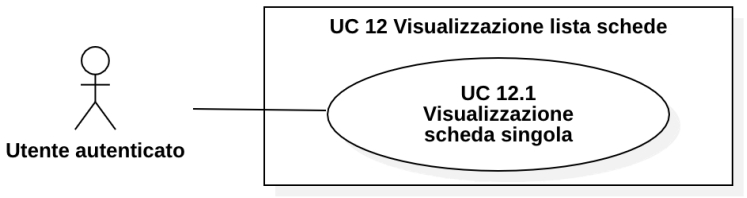


Figura 2.7: UC 12: Visualizzazione lista schede

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole visualizzare la lista delle schede.

Postcondizioni: L'utente ha visualizzato la lista delle schede.

Scenario principale:

- l'utente visualizza la lista delle schede.

Sottocasi:

- l'utente visualizza una scheda singola [UC 12.1].

UC 12.1: Visualizzazione scheda singola

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista delle schede.

Descrizione: L'utente vuole visualizzare una singola scheda.

Postcondizioni: L'utente ha visualizzato la singola scheda.

Scenario principale:

- l'utente visualizza la scheda singola.

Sottocasi:

1. l'utente visualizza il modello della scheda.
2. l'utente visualizza la descrizione della scheda.
3. l'utente visualizza il cliente della scheda.
4. l'utente visualizza la stagione della scheda.

UC 13: Visualizzazione dettaglio scheda

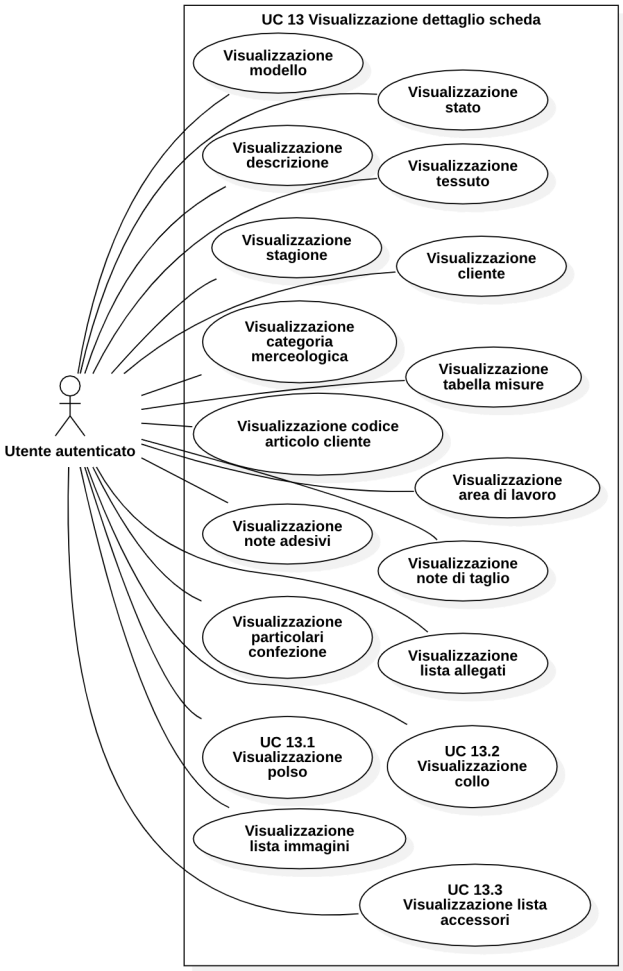


Figura 2.8: UC 13: Visualizzazione dettaglio scheda

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole visualizzare il dettaglio di una scheda.

Postcondizioni: L'utente ha visualizzato il dettaglio di una scheda.

Scenario principale:

- l'utente visualizza il modello;
- l'utente visualizza lo stato;
- l'utente visualizza la descrizione;
- l'utente visualizza il tessuto;
- l'utente visualizza la stagione;
- l'utente visualizza il cliente;
- l'utente visualizza la categoria merceologica;
- l'utente visualizza la tabella delle misure;
- l'utente visualizza il codice articolo cliente;
- l'utente visualizza l'area di lavoro;
- l'utente visualizza le note degli adesivi;
- l'utente visualizza le note di taglio;
- l'utente visualizza i particolari di confezione;
- l'utente visualizza il polso UC 13.1;
- l'utente visualizza il collo UC 13.2;
- l'utente visualizza gli allegati;
- l'utente visualizza le immagini;
- l'utente visualizza gli accessori [UC 13.3].

UC 13.1: Visualizzazione polso scheda

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato il dettaglio di una scheda.

Descrizione: L'utente vuole visualizzare il polso di una scheda.

Postcondizioni: L'utente ha visualizzato il polso di una scheda.

Scenario principale:

- l'utente visualizza il codice;
- l'utente visualizza la fustella;
- l'utente visualizza l'adesivo;
- l'utente visualizza l'impuntura;
- l'utente visualizza l'abbasso;
- l'utente visualizza la descrizione;
- l'utente visualizza le note.

UC 13.2: Visualizzazione collo scheda

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato il dettaglio di una scheda.

Descrizione: L'utente vuole visualizzare il collo di una scheda.

Postcondizioni: L'utente ha visualizzato il collo di una scheda.

Scenario principale:

- l'utente visualizza il codice;
- l'utente visualizza la spaziatura;
- l'utente visualizza le note;
- l'utente visualizza la fustella vela;
- l'utente visualizza l'adesivo collo;
- l'utente visualizza il rinforzo;
- l'utente visualizza la fustella fascetta;
- l'utente visualizza l'adesivo fascetta;
- l'utente visualizza la cucitura interna;
- l'utente visualizza l'impuntura vela;
- l'utente visualizza le note vela;
- l'utente visualizza l'impuntura fascetta;
- l'utente visualizza l'abbasso fascetta;
- l'utente visualizza l'impuntura naselli.
- l'utente visualizza le note fascetta.

UC 13.3: Visualizzazione lista accessori

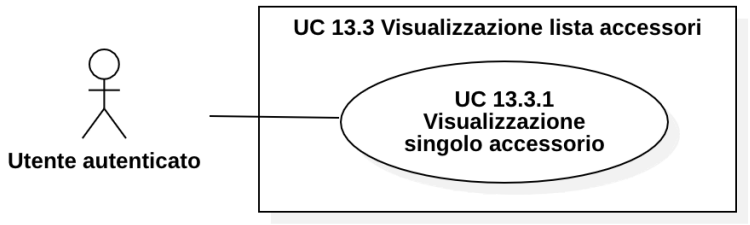


Figura 2.9: UC 13.3: Visualizzazione singolo accesorio

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato il dettaglio di una scheda.

Descrizione: L'utente vuole visualizzare gli accessori di una scheda.

Postcondizioni: L'utente ha visualizzato gli accessori di una scheda.

Scenario principale:

- l'utente visualizza la lista degli accessori.

Sottocasi:

- l'utente visualizza un accessorio singolo [UC 13.3.1].

UC 13.3.1: Visualizzazione accessorio singolo

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista degli accessori di una scheda.

Descrizione: L'utente vuole visualizzare un singolo accessorio di una scheda.

Postcondizioni: L'utente ha visualizzato il singolo accessorio di una scheda.

Scenario principale:

- l'utente visualizza il progressivo;
- l'utente visualizza il tipo;
- l'utente visualizza la dimensione;
- l'utente visualizza la posizione;
- l'utente visualizza la quantità;
- l'utente visualizza la quantità per la scorta.

UC 14: Modifica scheda

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole modificare una scheda.

Postcondizioni: L'utente ha modificato una scheda.

Scenario principale:

- l'utente modifica il modello;
- l'utente modifica lo stato;
- l'utente modifica la descrizione;
- l'utente modifica il tessuto;
- l'utente modifica la stagione;
- l'utente modifica il cliente;
- l'utente modifica la categoria merceologica;
- l'utente modifica la tabella delle misure;
- l'utente modifica il codice articolo cliente;
- l'utente modifica l'area di lavoro;

- l'utente modifica le note degli adesivi;
- l'utente modifica le note di taglio;
- l'utente modifica i particolari di confezione;
- l'utente modifica il polso;
- l'utente modifica il collo.

UC 15: Stampa scheda

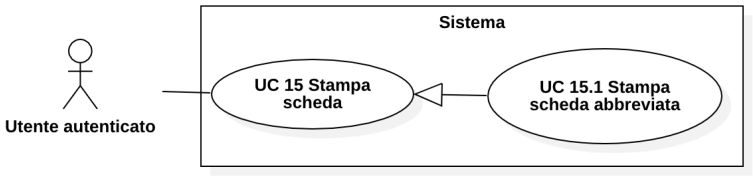


Figura 2.10: UC 15: Stampa scheda

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente almeno una scheda.

Descrizione: L'utente vuole stampare una scheda.

Postcondizioni: L'utente ha stampato una scheda.

Scenario principale:

- l'utente stampa una scheda.

Generalizzazioni:

- l'utente stampa la scheda in versione abbreviata [UC 15.1].

UC 15.1: Stampa scheda abbreviata

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente almeno una scheda.

Descrizione: L'utente vuole stampare una scheda in versione abbreviata.

Postcondizioni: L'utente ha stampato una scheda in versione abbreviata.

Scenario principale:

- l'utente stampa una scheda in versione abbreviata.

UC 16: Filtraggio schede

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista delle schede.

Descrizione: L'utente vuole filtrare la lista delle schede.

Postcondizioni: L'utente ha filtrato la lista delle schede.

Sottocasi:

- l'utente seleziona il cliente;
- l'utente seleziona la stagione;
- l'utente seleziona la categoria merceologica;
- l'utente seleziona la collezione;
- l'utente seleziona la tabella misure;
- l'utente seleziona lo stato;
- l'utente seleziona il polso;
- l'utente seleziona il collo;
- l'utente inserisce il modello da cercare.

UC 17: Eliminazione scheda

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole eliminare una scheda.

Postcondizioni: L'utente ha eliminato una scheda.

Scenario principale:

- l'utente elimina una scheda.

2.3.4 UC 18 - 31: Gestione di note, allegati, immagini e accessori

In questa sezione sono trattati gli use case relativi alle operazioni di gestione delle liste trattate da un cliente o da una scheda. Viste le due liste in comune (allegati e immagini) sono stati unificati gli use case in questione senza fare differenze tra cliente e scheda.
La visualizzazione non è presente in quanto è già trattata in UC 6 e UC 13.

UC 18: Inserimento nota

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente.

Descrizione: L'utente vuole inserire una nota.

Postcondizioni: L'utente ha inserito una nota.

Scenario principale:

- l'utente inserisce il contenuto.

UC 19: Modifica nota

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente che possiede una nota.

Descrizione: L'utente vuole modificare una nota.

Postcondizioni: L'utente ha modificato una nota.

Scenario principale:

- l'utente modifica il contenuto.

UC 20: Eliminazione nota

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente che possiede una nota.

Descrizione: L'utente vuole eliminare una nota.

Postcondizioni: L'utente ha eliminato una nota.

Scenario principale:

- l'utente elimina una nota.

UC 21: Inserimento allegato

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente o una scheda.

Descrizione: L'utente vuole inserire un allegato.

Postcondizioni: L'utente ha inserito un allegato.

Scenario principale:

- l'utente seleziona il tipo;
- l'utente inserisce la descrizione;
- l'utente inserisce il file.

UC 22: Modifica descrizione allegato

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente o una scheda che possiede un allegato.

Descrizione: L'utente vuole modificare un allegato.

Postcondizioni: L'utente ha modificato un allegato.

Scenario principale:

- l'utente modifica la descrizione.

UC 23: Download allegato

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente o una scheda che possiede un allegato.

Descrizione: L'utente vuole scaricare un allegato.

Postcondizioni: L'utente ha scaricato un allegato.

Scenario principale:

- l'utente scarica un allegato.

UC 24: Eliminazione allegato

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente o una scheda che possiede un allegato.

Descrizione: L'utente vuole eliminare un allegato.

Postcondizioni: L'utente ha eliminato un allegato.

Scenario principale:

- l'utente elimina un allegato.

UC 25: Inserimento immagine

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente o una scheda.

Descrizione: L'utente vuole inserire un'immagine.

Postcondizioni: L'utente ha inserito un'immagine.

Scenario principale:

- l'utente seleziona il tipo;
- l'utente inserisce la descrizione;
- l'utente inserisce il file.

UC 26: Modifica descrizione immagine

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente o una scheda che possiede un'immagine.

Descrizione: L'utente vuole modificare un'immagine.

Postcondizioni: L'utente ha modificato un'immagine.

Scenario principale:

- l'utente modifica la descrizione.

UC 27: Download immagine

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente o una scheda che possiede un'immagine.

Descrizione: L'utente vuole scaricare un'immagine.

Postcondizioni: L'utente ha scaricato un'immagine.

Scenario principale:

- l'utente scarica un'immagine.

UC 28: Eliminazione immagine

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente un cliente o una scheda che possiede un'immagine.

Descrizione: L'utente vuole eliminare un'immagine.

Postcondizioni: L'utente ha eliminato un'immagine.

Scenario principale:

- l'utente elimina un'immagine.

UC 29: Inserimento accessorio

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente una scheda.

Descrizione: L'utente vuole inserire un accessorio.

Postcondizioni: L'utente ha inserito un accessorio.

Scenario principale:

- l'utente inserisce il progressivo;
- l'utente inserisce il tipo;
- l'utente inserisce la dimensione;
- l'utente inserisce la posizione;
- l'utente inserisce la quantità;
- l'utente inserisce la quantità per la scorta.

UC 30: Modifica accessorio

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente una scheda che possiede un accessorio.

Descrizione: L'utente vuole modificare un accessorio.

Postcondizioni: L'utente ha modificato un accessorio.

Scenario principale:

- l'utente modifica il tipo;
- l'utente modifica la dimensione;
- l'utente modifica la posizione;
- l'utente modifica la quantità;
- l'utente modifica la quantità per la scorta.

UC 31: Eliminazione accessorio

Attori principali: Utente autenticato.

Precondizioni: Nel sistema è presente una scheda che possiede un accessorio.

Descrizione: L'utente vuole eliminare un accessorio.

Postcondizioni: L'utente ha eliminato un accessorio.

Scenario principale:

- l'utente elimina un accessorio.

2.3.5 UC 32 - 36: Gestione di attributi generici

Vista la complessità del sistema è stato deciso di gestire gli use case dei restati attributi della scheda in maniera generica. Infatti molti di essi richiedono la gestione di una lista di elementi, con la necessità delle operazioni di inserimento, modifica ed eliminazione.

UC 32: Inserimento attributo generico

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole inserire un elemento di una tabella.

Postcondizioni: L'utente ha inserito un elemento di una tabella.

Scenario principale:

- l'utente inserisce tutti i campi necessari per la creazione dell'elemento.

UC 33: Visualizzazione lista attributi generica

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole visualizzare la lista degli elementi di una tabella.

Postcondizioni: L'utente ha visualizzato la lista degli elementi di una tabella.

Scenario principale:

- l'utente visualizza la lista degli elementi di una tabella.

Sottocasi:

- l'utente visualizza un elemento singolo [UC 33.1].

UC 33.1: Visualizzazione attributo generico singolo

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista degli elementi di una tabella.

Descrizione: L'utente vuole visualizzare un singolo elemento di una tabella.

Postcondizioni: L'utente ha visualizzato un singolo elemento di una tabella.

Scenario principale:

- l'utente visualizza la descrizione dell'elemento.

UC 34: Modifica attributo generico

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole modificare un elemento di una tabella.

Postcondizioni: L'utente ha modificato un elemento di una tabella.

Scenario principale:

- l'utente modifica i campi desiderati dell'elemento.

UC 35: Filtraggio lista attributi generica

Attori principali: Utente autenticato.

Precondizioni: L'utente ha visualizzato la lista degli elementi di una tabella.

Descrizione: L'utente vuole filtrare la lista degli elementi di una tabella.

Postcondizioni: L'utente ha filtrato la lista degli elementi di una tabella.

Scenario principale:

- l'utente inserisce la descrizione dell'elemento da cercare.

UC 36: Eliminazione attributo generico

Attori principali: Utente autenticato.

Precondizioni: L'utente ha accesso al sistema.

Descrizione: L'utente vuole eliminare un elemento di una tabella.

Postcondizioni: L'utente ha eliminato un elemento di una tabella.

Scenario principale:

- l'utente elimina un elemento di una tabella.

2.3.6 UCE 1 - 2: Errori riguardanti l'inserimento di testo

Vista la complessità del sistema non sono stati elencati tutti gli specifici e minimi errori. Inoltre non sono presenti grossi errori che possono compromettere il sistema.

UCE1: Errore credenziali non valide

Attori principali: Utente non autenticato.

Precondizioni: L'utente ha inserito delle credenziali errate.

Descrizione: L'utente visualizza un errore che avvisa che le credenziali non sono corrette.

Postcondizioni: L'utente ha visualizzato l'errore e deve inserire di nuovo le credenziali.

UCE2: Errore campi non compilati

Attori principali: Utente non autenticato.

Precondizioni: L'utente non ha inserito tutti i campi obbligatori.

Descrizione: L'utente visualizza un errore che avvisa che ci sono campi obbligatori non compilati.

Postcondizioni: L'utente ha visualizzato l'errore e deve inserire i campi obbligatori mancanti.

2.4 Tracciamento requisiti

In seguito alla fase di stesura degli use case, sono stati individuati i requisiti del sistema.
Come per i casi d'uso anche i requisiti trattati da questo documento sono limitati vista la quantità e la somiglianza tra di essi.
Ogni requisito è identificato da un codice univoco costituito come segue R[Importanza][Tipologia][Progressivo]:

- **Importanza**: può assumere i seguenti valori:
  - **1**: requisito obbligatorio;
  - **2**: requisito desiderabile, ma non necessario;
  - **3**: requisito facoltativo, cioè utile ma valutabile in futuro.
- **Tipologia**: può assumere i seguenti valori:
  - **F**: requisito funzionale, cioè che specifica una funzione che il sistema deve soddisfare;
  - **Q**: requisito qualitativo, cioè che specifica una proprietà che il sistema deve possedere;
  - **P**: requisito prestazionale, cioè che il sistema deve avere una determinata prestazione;
  - **V**: requisito di vincolo.
- **Progressivo**: numero progressivo.

Le tabelle saranno formate da 3 colonne che indicano:

- **Requisito**: codice identificativo del requisito;
- **Descrizione**: breve descrizione del requisito;
- **Fonte**: fonte del requisito, che può essere uno use case oppure *interno*, cioè identificato dal sottoscritto.

2.4.1 Requisiti funzionali

Requisito	Descrizione	Fonte
R1F1	Un utente deve poter effettuare il login	UC 1
R1F2	Un utente deve poter effettuare il logout	UC 2
R1F3	Il sistema deve permettere l'inserimento di un cliente	UC 4
R1F4	Il sistema deve permettere la visualizzazione dei clienti	UC 5
R1F5	Il sistema deve permettere la visualizzazione dei dettagli di un cliente	UC 6
R1F6	Il sistema deve permettere la visualizzazione delle note di un cliente	UC 6.1
R1F7	Il sistema deve permettere la visualizzazione degli allegati di un cliente o di una scheda	UC 6.2, Allegati scheda
R1F8	Il sistema deve permettere la visualizzazione delle immagini di un cliente o di una scheda	UC 6.2, Immagini scheda
R1F9	Il sistema deve permettere la modifica di un cliente	UC 7
R1F10	Il sistema deve permettere l'attivazione e la disattivazione di un cliente	UC 8
R1F11	Il sistema deve permettere il filtraggio dei clienti	UC 9
R1F12	Il sistema deve permettere l'eliminazione di un cliente	UC 10
R1F13	Il sistema deve permettere l'inserimento di una scheda	UC 11
R1F14	Il sistema deve permettere la visualizzazione di tutte le shede	UC 12
R1F15	Il sistema deve permettere la visualizzazione dei dettagli di una scheda	UC 13
R1F16	Il sistema deve permettere la visualizzazione degli accessori di una scheda	UC 13.3
R1F17	Il sistema deve permettere la modifica di una scheda	UC 14

Requisito	Descrizione	Fonte
R1F18	Il sistema deve permettere la stampa di una scheda	UC 15
R1F19	Il sistema deve permettere il filtraggio delle schede	UC 16
R1F20	Il sistema deve permettere l'eliminazione di una scheda	UC 17
R1F21	Il sistema deve permettere l'inserimento di una nota	UC 18
R1F22	Il sistema deve permettere la modifica di una nota	UC 19
R1F23	Il sistema deve permettere l'eliminazione di una nota	UC 20
R1F24	Il sistema deve permettere l'inserimento di un allegato	UC 21
R1F25	Il sistema deve permettere la modifica di un allegato	UC 22
R1F26	Il sistema deve permettere il download di un allegato	UC 23
R1F27	Il sistema deve permettere l'eliminazione di un allegato	UC 24
R1F28	Il sistema deve permettere l'inserimento di un'immagine	UC 25
R1F29	Il sistema deve permettere la modifica di un'immagine	UC 26
R1F30	Il sistema deve permettere il download di un'immagine	UC 27
R1F31	Il sistema deve permettere l'eliminazione di un'immagine	UC 28
R1F32	Il sistema deve permettere l'inserimento di un accessorio	UC 29
R1F33	Il sistema deve permettere la modifica di un accessorio	UC 30
R1F34	Il sistema deve permettere l'eliminazione di un accessorio	UC 31
R1F35	Il sistema deve permettere l'inserimento di un attributo generico della scheda	UC 32
R1F36	Il sistema deve permettere la visualizzazione della lista di attributi generici della scheda	UC 33
R1F37	Il sistema deve permettere la modifica di un attributo generico della scheda	UC 34
R1F37	Il sistema deve permettere il filtraggio di liste di attributi generici della scheda	UC 35
R1F38	Il sistema deve permettere l'eliminazione di un attributo generico della scheda	UC 36
R2F1	Il sistema deve poter permettere la stampa semplificata di una scheda prodotto	UC 15.1
R3F1	Il sistema deve permettere il cambio della lingua	UC 3

Tabella 2.1: Tabella dei requisiti funzionali

2.4.2 Requisiti qualitativi

Requisito	Descrizione	Fonte
R1Q1	Il sistema dev'essere utilizzabile in diversi dispositivi e scalare correttamente in base alle dimensioni dello schermo, in sintesi responsive	Interno
R1Q2	Il codice del sistema dev'essere caricato in due repository (frontend e backend) di DevOps	Interno

Tabella 2.2: Tabella dei requisiti qualitativi

2.4.3 Requisiti di vincolo

Requisito	Descrizione	Fonte
R1V1	Il backend dev'essere sviluppato in C# secondo una specifica architettura approfondita nel capitolo di progettazione	Interno
R1V2	Il frontend dev'essere sviluppato in TypeScript e Angular secondo una specifica architettura approfondita nel capitolo di progettazione	Interno
R1V3	Il frontend deve utilizzare Bootstrap	Interno
R1V4	I template delle schede devono essere creati tramite Jasper	Interno
R1V5	Frontend e backend devono eseguire su container Docker	Interno
R1V6	I dati vanni persistiti su un database	Interno

Tabella 2.3: Tabella dei requisiti di vincolo

2.4.4 Requisiti prestazionali

Non ci sono stati particolari requisiti prestazionali richiesti dal cliente. Inoltre il sistema non deve gestire grandi moli di dati o complesse operazioni, quindi si comporta in modo reattivo rispetto ai bisogni che gli utenti hanno.

Capitolo 3

Progettazione

3.1 Domain-Driven Design

La decisione di utilizzare il Domain-Driven Design (successivamente nominato per brevità *DDD*) ha comportato che la concentrazione iniziale fosse principalmente nell'individuare come si componesse il dominio del sistema. Nel dominio vanno definiti una serie di oggetti, che sono organizzati tramite i seguenti tipi:

- **entità**: sono oggetti che hanno una storia (vengono create, inserite, si evolvono...) e un identità, cioè anche se esiste un altro oggetto con attributi identici, si può distinguere l'uno dall'altro;
- **value object**: sono oggetti che vengono definiti esclusivamente dal loro valore e sono immutabili; dove è possibile è meglio utilizzarli, perché sono meno complessi di un'entità;
- **aggregate**: sono insiemi di entità e value object che singolarmente non garantiscono l'integrità del dominio, ma se messi insieme sono consistenti; ogni aggregate possiede un rappresentante, detto **aggregate root**: sarà lui a gestire le interazioni con l'esterno e a garantire il controllo delle invarianti; tutte le entità restanti vengono definite **entità figlie**: dall'esterno non ci si può accedere.

Il primo passo è stato quindi quello di definire il dominio del sistema: l'azienda ospitante aveva già effettuato un'analisi preliminare con il cliente, quindi è stato relativamente semplice individuare entità, value object e aggregate. Inoltre durante lo sviluppo il dominio è stato rivisto in seguito ad aggiornamenti con il cliente e a nuove esigenze emerse.

Il DDD non ha standard a riguardo della rappresentazione del dominio: è stato quindi rappresentato per semplicità tramite gli strumenti dei diagrammi delle classi. Nelle figure successive sono rappresentati tutti gli aggregate root (AR), entità figlie (CE) e value object (VO) individuati, con le frecce che puntano a un AR partendo dagli oggetti figli. Viene anche indicato il tipo degli attributi che rappresentano i vari oggetti, con eventuali *'?'* per indicare che alcuni attributi possono essere nulli.

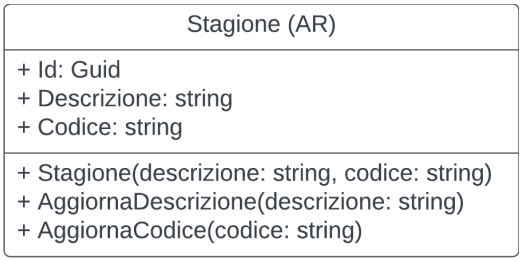


Figura 3.1: Aggregato Stagione

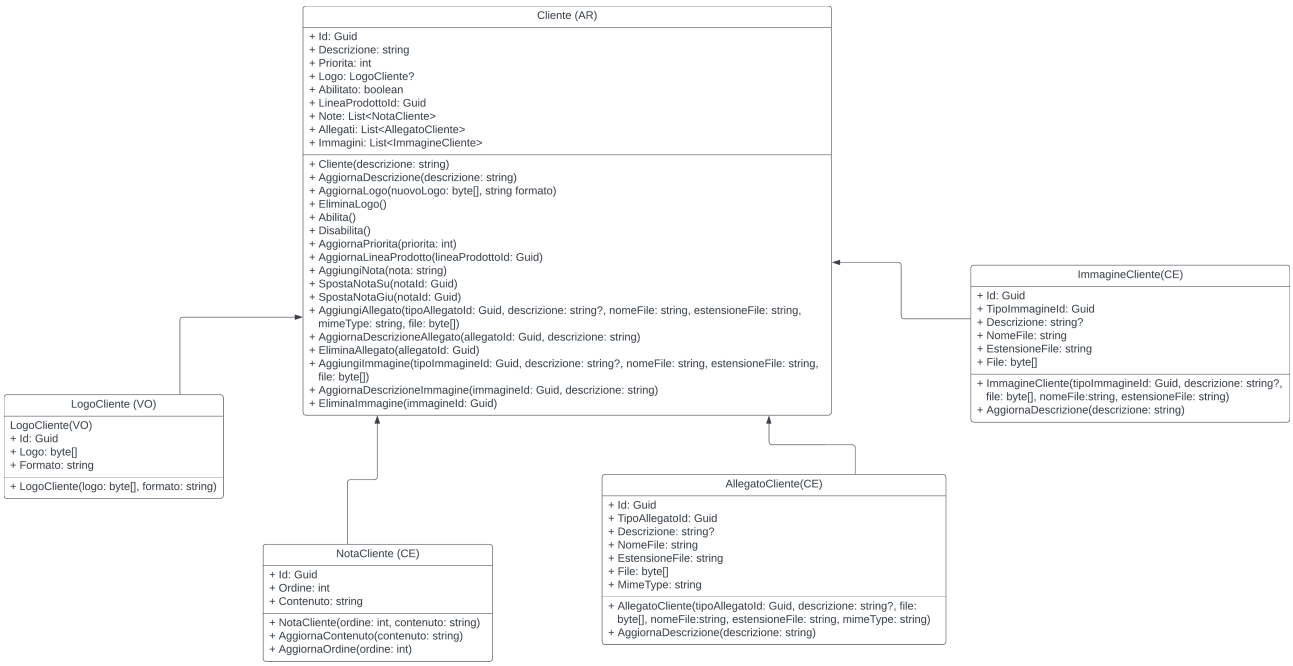


Figura 3.2: Aggregato Cliente

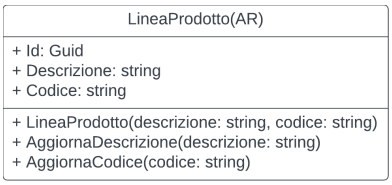


Figura 3.3: Aggregato LineaProdotto

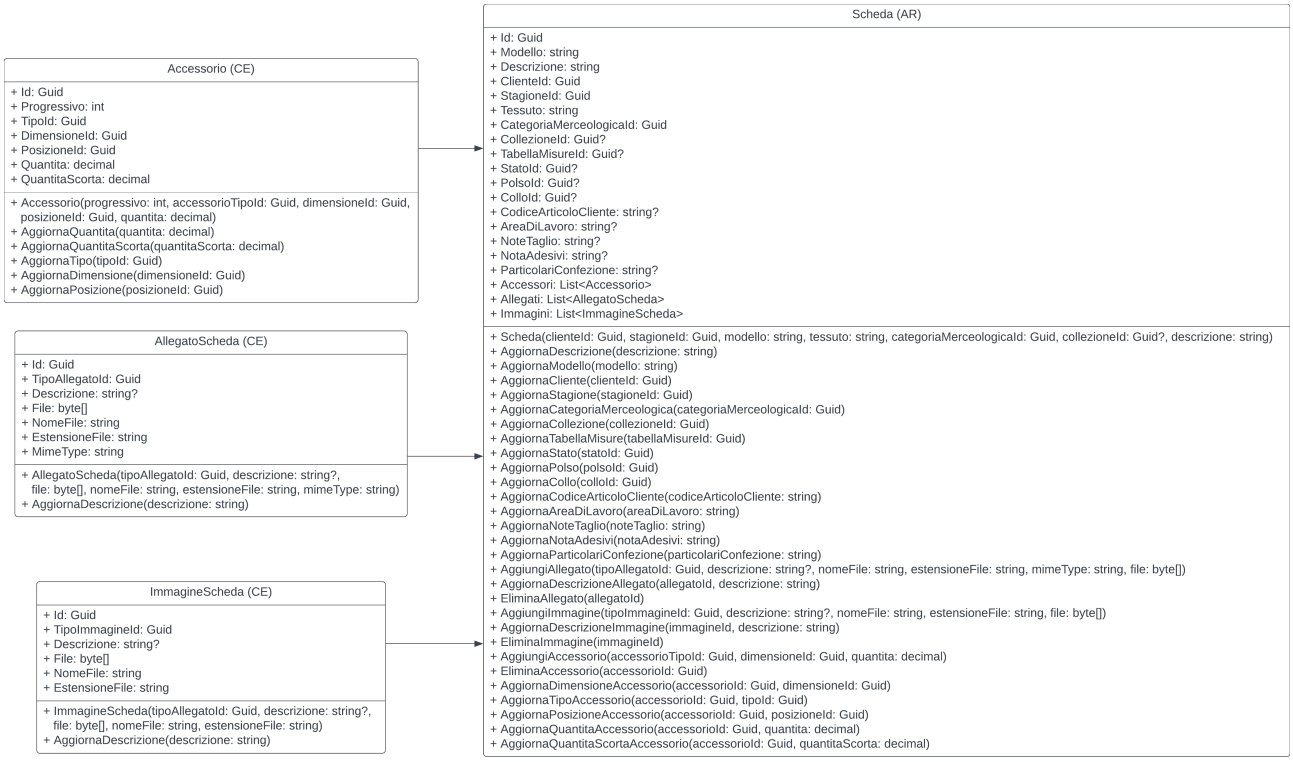


Figura 3.4: Aggregato Scheda

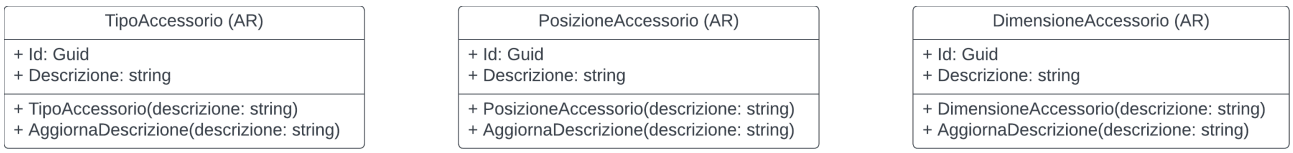
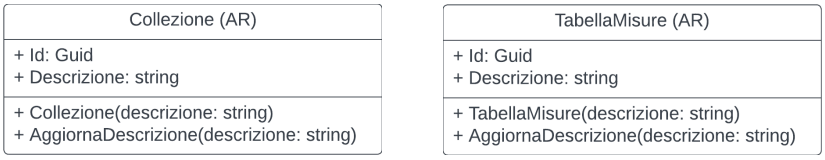


Figura 3.5: Aggregati riguardanti l'accessorio



(a) Aggregato Collezione **(b)** Aggregato TabellaMisure

Figura 3.6: Aggregati Collezione e TabellaMisure

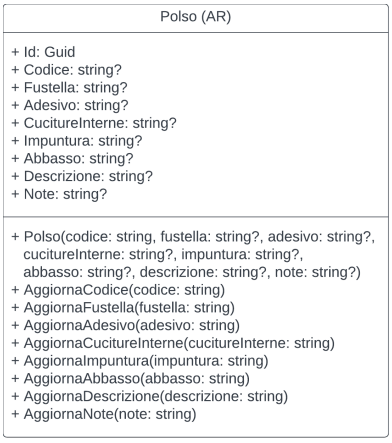


Figura 3.7: Aggregato Polso

Figura 3.8: Aggregato Collo

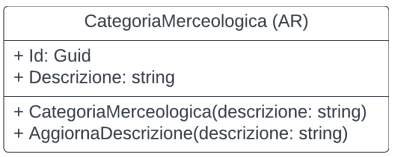


Figura 3.9: Aggregato CategoriaMerceologica

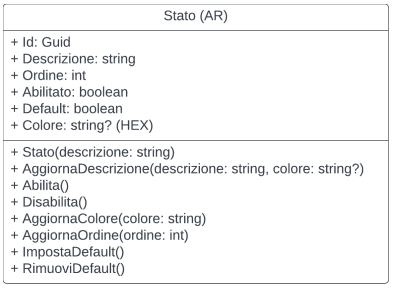


Figura 3.10: Aggregato Stato

Questo è stato il primo dominio definito. Come detto inizialmente, durante lo sviluppo e tuttora è in aggiornamento continuo.

3.2 Schema generale

L'architettura generale del sistema mi è stata indicata dall'azienda sulla base di quelle che sono già utilizzate per altri progetti. In seguito in figura lo schema generale del sistema.

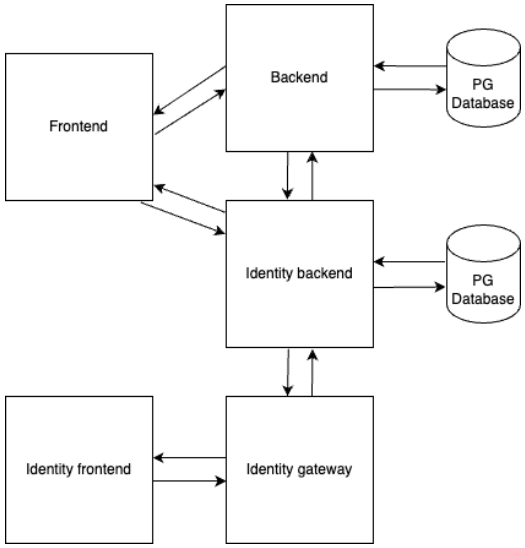


Figura 3.11: Schema generale

Le parti oggetto di tirocinio sono state principalmente frontend e backend. Sono due applicazioni eseguite su container docker separati che comunicano tra loro tramite API REST. Il backend inoltre interagisce con un database relazionale PostgreSQL, utilizzato per la persistenza dei dati. La parte di identity invece gestisce autenticazione e autorizzazione degli utenti. Non sarà discussa in questo documento in quanto era già stata sviluppata in precedenza. Verrà trattata solo l'unica interazione che essa ha attualmente con il sistema sviluppato, cioè il login nel frontend.

3.3 Frontend

Il frontend è stato sviluppato tramite Angular, un framework open-source per la creazione di applicazioni web eseguibili in tutti i browser più diffusi. Il client quindi quando dovrà eseguire la webapp scaricherà il codice dal server e lo eseguirà in locale tramite browser. Angular cerca di rendere lo sviluppo indipendente dal dispositivo, in modo da non dover variare eccessivamente lo stile della grafica in base alla dimensione e alla risoluzione dello schermo da cui vi si accede.

Figura 3.12: Angular

Per costruire un applicazione esso utilizza diversi elementi.
L'elemento fondamentale è il *componente* che idealmente dovrebbe occuparsi solo delle questioni grafiche; è costituito minimo da tre parti:

- **template**: scritta in HTML, è la parte da rappresentare graficamente all'utente;
- **classe**: scritta in TypeScript, è la parte che definisce il comportamento del componente;
- **selettore di stile**: è la parte che definisce lo stile del componente e può avere diversi formati, in questo caso è stato utilizzato SCSS, un superset di CSS.

(a) HTML **(b)** TypeScript **(c)** SCSS

Figura 3.13: Linguaggi utilizzati da un componente Angular

Un altro elemento utilizzato è il *servizio*, che è una classe TypeScript con uno scopo preciso e ben definito, utilizzato dai componenti per svolgere operazioni non relazionate alla grafica: ad esempio chiamate alle API, validazione dei dati, ecc.
Ci sono poi i *moduli*, sempre classi TypeScript che vanno a gestire i componenti, i servizi e le altre classi che compongono l'applicazione.
Un altro aspetto fondamentale di Angular è la *dependency injection*, cioè un tecnica che consiste nel fornire a un oggetto le istanze di una classe di cui ha bisogno senza che se le debba creare da solo. Questo permette di avere meno accoppiamento tra una classe e le sue dipendenze, inoltre rende il codice facilmente testabile e riusabile. Angular utilizza questo meccanismo per fornire ai componenti eventuali servizi o moduli di cui hanno bisogno.
Per rendere più accattivanti le schermate dell'applicazione web sono state utilizzate diverse librerie grafiche che forniscono elementi e stili già pronti da utilizzare.
La principale è stata **Bootstrap**, una libreria open source gratuita che contiene una serie di componenti sviluppati in HTML e CSS per la creazione di siti e applicazioni web. In particolare assieme ad Angular ne escono risultati responsive.
Un'altra libreria utilizzata è stata **Kendo**, sempre una libreria grafica, utilizzata soprattutto per la creazione

di tabelle e per alcuni form.
Infine per i tasti e le icone è stata utilizzata la libreria **Font Awesome**, un toolkit che gratuitamente (o a pagamento per sbloccarne di più) fornisce una serie di immagini utilizzabili tramite un tag HTML.

(a) Bootstrap **(b)** Kendo **(c)** Font Awesome

Figura 3.14: Librerie del frontend

Il frontend è stato organizzando secondo una struttura definita dall'azienda. Il progetto è diviso in due parti, una generale che va a descrivere tutti i componenti e i moduli comuni e una specifica per il cliente, che contiene tutte le parti specifiche richieste: in questo modo è possibile riutilizzare il codice generale nel caso di bisogno per altri clienti.
La maggior parte del lavoro è stata svolta nella parte generale: non avendo altri clienti attuali non era necessario avere una parte specifica. Al suo interno i componenti erano organizzati principalmente secondo il dominio, cioè in base agli *aggregate*: c'è in genere una cartella per ogni aggregate root. Ogni cartella possiede almeno:

- il modulo che dichiara e gestisce il routing di tutti i sottocomponenti della cartella e le eventuali dipendenze da iniettare;
- un componente con il compito di mostrare la lista degli elementi del tipo dell'aggregate root, utilizzando quasi sempre una Kendo Grid.

Poi in caso di necessità ci sono anche altri componenti: ad esempio se c'è troppa complessità in un componente lo si può dividere in diversi componenti detti *"figli"*, oppure se c'è il bisogno di un [G]modale viene creato un componente apposito.

3.4 Backend

Il backend è un insieme di progetti C# sviluppati tramite *.NET*, framework open source per la creazione di molteplici tipi di applicazioni, e una libreria sviluppata internamente dall'azienda ospitante. Essa contiene tutto il codice necessario per esporre le chiamate API, gestire la comunicazione con il database e gestire le autorizzazioni.

(a) .NET **(b)** C#

Figura 3.15: Strumenti del backend

3.4.1 Pattern e metodologie

Prima di approfondire la struttura vanno definiti alcuni concetti per la più chiara comprensione del sistema.
Il primo è il pattern detto **DTO** (Data Transfer Object). Un DTO è una rappresentazione di un oggetto utilizzata per la comunicazione tra due applicazioni software o tra due componenti di un'applicazione. In genere è utilizzato come oggetto in cui verranno mappati i dati ottenuti da un *DAO* (Data Access Object), cioè la rappresentazione di ciò che è stato ottenuto da un database. Sono stati utilizzati principalmente per questo scopo e per la definizione degli svariati payload delle API.
Un altro pattern utilizzato è stato il **CQRS** (*Command and Query Responsibility Segregation*), che consiste nel dividere le richieste che vanno a effettuare operazioni di scrittura di dati (indipendentemente se nella memoria o nella base di dati) da quelle che invece leggono i dati. Questo modello permette di avere [G] query più semplici, scalabilità indipendente e maggiore sicurezza.
È stato utilizzato anche il pattern **Repository**: consiste semplicemente nel gestire le operazioni di una collezione (inserimento, lettura, modifica...) tramite un'interfaccia, in modo che ci sia meno accoppiamento. Questo pattern utilizzato in particolare con il DDD permette di mantenere gli oggetti del dominio *"ignoranti"* da eventuali operazioni effettuate con database.
Inoltre la libreria interna permette anche l'utilizzo del pattern **Unit of Work**. L'idea è che se devono essere effettuate molteplici operazioni su un database, bisogna eseguirle tutte oppure nessuna, in modo da non riportare nel database dati incompleti.

3.4.2 Database

Il database è interamente gestito da **EFCore** (*Entity Framework Core*), un framework **ORM** (*Object Relational Mapping*) sviluppato per .NET. Un ORM facilita le interazioni tra i linguaggi di programmazione orientati agli oggetti e i database. In questo caso è stato adottato l'approccio *Code First*, quello che a cui EFCore fa più riferimento: utile inoltre per il DDD. Questo approccio consiste nel far creare il database e le tabelle dalle API del framework, in base a ciò che è stato definito nel codice. Funziona attraverso migration, cioè classi che descrivono quali modifiche effettuare nel database.
Non è stato riportato un vero e proprio [G]diagramma E-R, in quanto in applicazioni semplici nel DDD il dominio è in corrispondenza 1:1 con il database.

Figura 3.16: EFCore

3.4.3 API

Le API sono state progettate cercando di seguire il più possibile i principi REST. In genere ogni entità del dominio ha le API necessarie a eseguire le stesse operazioni: inserimento, modifica, eliminazione, lettura e filtraggio di tutti gli elementi della tabella.
Tra l'altro, grazie alla libreria interna dell'azienda ospitante, c'era una creazione automatica di una pagina web *Swagger* che definisce tutte le API e permette di testarle manualmente.

3.4.4 Struttura

L'esecuzione parte da un progetto denominato **host**, che contiene tutte le impostazioni e il file di esecuzione principale. I restanti progetti sono divisi in 2 parti, *core* e *infrastructure*.
La parte **core** è composta da cinque progetti:

- **ApplicationService**: contiene il codice per la gestione delle API; si occupa ad esempio dell'avvio del servizio, di tutti gli handler che gestiscono le richieste effettuate e della creazione dei report;
- **Commmand**: come descritto nel CQRS contiene tutti i comandi e i rispettivi handler, cioè i responsabili di gestire operazioni di inserimento/modifica su database;
- **Domain**: qui sono modellati gli oggetti definiti nel dominio; sono anche definite le interfacce delle repository, come descritto nel pattern repository; qui viene gestita tutta la logica di business;
- **DTO**: in questa parte ci sono tutti i file che vanno a definire i DTO, utilizzati per la comunicazione all'interno del sistema e con l'esterno;
- **Query**: in questo progetto, per la divisione da eseguire secondo il CQRS, c'è il codice che va a descrivere le operazioni di lettura su database.

La parte **infrastructure** è la parte responsabile dell'interazione con il database. È questa parte che va ad effettuare veramente le operazioni richiamate dai comandi e dalle query. È divisa in due cartelle, sempre secondo la logica del CQRS, denominate *Persistence* e *Query*. Ognuna contiene un progetto che va a definire il comportamento comune a tutti i database, e un progetto per ogni database utilizzato. In questo caso si tratta solo di un progetto dedicato a PostgreSQL, ma se fosse necessario l'utilizzo di un database di diverso tipo in

futuro basterebbe aggiungere un nuovo progetto, quasi senza modificare il codice già esistente.

- Cartella **Persistence**: il progetto generale contiene principalmente le vere implementazioni dei repository (pattern repository); il progetto specifico per PostgreSQL contiene le migration di EFCore, la gestione del pattern Unit of Work e le configurazioni specifiche per il database. Inoltre in entrambi i progetti ci sono le configurazioni delle classi del dominio per l'utilizzo di EFCore.
- Cartella **Query**: nel progetto generale ci sono gli handler che gestiscono le vere e proprie operazioni di lettura su database, le configurazioni per il mapping DAO-DTO tramite la libreria Automapper e la definizione dei DAO. Come nella cartella di persistence in entrambi i progetti ci sono le configurazioni per EFCore, ma in questo caso del modello di lettura.

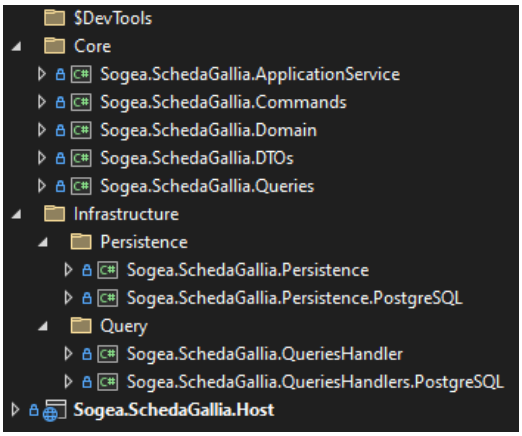


Figura 3.17: Strutta dei file del backend

Capitolo 4

Realizzazione

4.1 Metodo di lavoro

Durante lo sviluppo è stato richiesto dall'azienda di seguire delle regole riguardanti il metodo di lavoro da utilizzare durante lo sviluppo del sistema: esse erano le stesse seguite durante i progetti aziendali, in modo da avere un esperienza più vicina possibile a quella lavorativa.
In questa sezione verranno descritte le metodologie utilizzate per quanto riguarda il versionamento, la revisione del codice e il tracciamento delle attività da svolgere.

4.1.1 Versionamento e code review

Ci sono state regole per il versionamento, in modo da poter avere un controllo sulle modifiche effettuate e per poterle ripristinare in caso di errori anche in mia assenza o dopo la conclusione del periodo di tirocinio.
I repository utilizzati erano entrambi ospitati su cloud da Microsoft Azure Repos, il servizio di hosting offerto da Azure DevOps.
Come software di versione di controllo è stato utilizzato **Git**. Per semplificare e velocizzare le operazioni sono stati utilizzati due client, Fork per il frontend e il client di default installato in Visual Studio per il backend.
Doveva essere seguito un workflow specifico per il versionamento, che prevede l'utilizzo di questi branch:

- **main**: è il branch utilizzato per la creazione iniziale del progetto, inutilizzato durante tutto il resto dello sviluppo;
- **feature/nome-feature**: sono i branch dedicati allo sviluppo vero e proprio, ognuno deve implementare una funzionalità specifica;
- **dev**: viene inizialmente creato dal main; è il principale branch di sviluppo;
- **test/nome-azienda**: ogni azienda ha un branch dedicato; quando ci sono nuovi sviluppi e ci sono da fare test a riguardo, viene effettuato il merge da dev a questo branch;
- **release/nome-azienda**: ogni azienda ha un branch dedicato; quando i test sono terminati e va rilasciata una versione al cliente viene effettuato il merge da test a questo branch.

Per ogni nuova funzionalità veniva creato un nuovo branch feature: esso partiva dal branch dev o da un altro branch feature, a seconda della situazione. Nel caso in cui non ci fossero stati altri branch feature in sviluppo, partiva da dev. Altrimenti da un altro branch feature, in genere quando il branch da cui veniva creato era in fase di code review.
In questo caso i branch di test e di release erano sempre singoli, visto che come detto nei capitoli precedenti questo progetto era dedicato a un unico cliente.
In seguito a una pull request di un branch feature il merge in dev non era automatico, il codice era sottoposto a una procedura di code review da parte del tutor aziendale. In questo modo la qualità del codice è potuta rimanere di un livello accettabile per gli standard aziendali, evitando di introdurre complessità che sarebbero potute emergere in futuro. Anche questo era totalmente gestito tramite Azure DevOps.

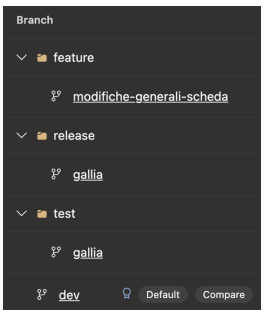


Figura 4.1: Branch visualizzati in Azure DevOps

4.1.2 Boards

Per il tracciamento delle attività da svolgere è stato utilizzato il servizio di Azure DevOps Boards. Veniva seguito il framework agile Scrum con sprint da due settimane. Non sono state seguite completamente tutte le regole della metodologia in quanto essendoci un unico sviluppatore il lavoro era abbastanza autonomo, senza interferenze esterne.
All'interno di ogni sprint venivano inseriti i **PBI** che dovevano essere svolti, con al loro interno i *task* che li componevano, oppure **bug** riscontrati durante lo sviluppo.
A ogni pull request venivano associati i vari task che erano stati svolti o i bug risolti durante lo sviluppo di quel determinato branch. Quando veniva approvata e DevOps effettuava il merge, i task venivano automaticamente marcati come completati. In seguito, quando tutti i task di un PBI erano terminati, anche questo veniva marcato come completato manualmente dal tutor.

4.2 Frontend

4.2.1 Servizi e chiamate delle API

Tutta la parte di comunicazione con il backend è gestita tramite delle classi denominate **servizi**. Queste sono marcate dal decoratore *@Injectable*, per poter essere iniettate in altri componenti. Ogni metodo della classe va a definire una chiamata API.
Per effettuare le chiamate è stato sfruttato un servizio già creato dall'azienda ospitante, che a sua volta utilizza la libreria HTTP standard di Angular. Nell'immagine 4.3 si possono notare infatti i metodi *getCustomer()* e *addCustomer()*, che appunto fanno una chiamata HTTP al backend.
Gli argomenti dei metodi per le chiamate sono gli eventuali parametri e payload della richiesta da inviare; il tipo di ritorno invece è sempre un Observable di un certo tipo definito tramite un DTO. Per fare le chiamate API è stato necessario l'utilizzo della libreria *RxJS*. Questa è una libreria in JavaScript per la programmazione reattiva: utilizza gli *observable* per poter gestire chiamate asincrone. Di seguito in esempio c'è una chiamata al servizio dei clienti, con la relativa gestione tramite RxJS dell'observable ritornato.

```
this.customerSvc.getEnabledCustomers()
  .pipe(handleLoading(this))
  .subscribe({
    next: x => this.customers = x.values,
    error: err => this.errorSvc.handle(err)
  });
```

Figura 4.2: Chiamata che ottiene i clienti attivi

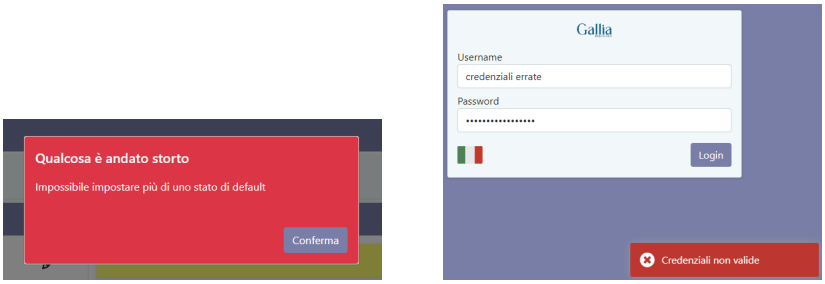
Altri metodi caratteristici di questi servizi sono *getGridData()* (un nome generico che richiama semplicemente il metodo che ritorna la lista) e *getDefaultFilter()* (che ritorna il filtro di base con cui fare la chiamata per la lista), utilizzati per l'ottenimento di liste di oggetti. In particolare sono stati utilizzati per la gestione della paginazione delle tabelle.

```
@Injectable()
export class CustomerService extends BaseFilterService<CustomerFilter, CustomerListItem> {
  public readonly url = 'sogea.schedagallia.service/customers';

  public constructor(private httpSvc: PermissionsHttpService) {
    super();
  }

  public getGridData(): Observable<BaseList<CustomerListItem>> {
    return this.getCustomers();
  }

  public getDefaultFilter(): Readonly<CustomerFilter> {
    return defaultGetListCustomer;
  }

  public getCustomers(filter?: CustomerFilter): Observable<BaseList<CustomerListItem>> {
    filter = filter || this.filterValue;
    return this.httpSvc.post<BaseList<CustomerListItem>>(`${this.url}/GetCustomerList`, filter);
  }

  public getCustomer(id: string): Observable<Customer> {
    return this.httpSvc.get<Customer>(`${this.url}/${id}`);
  }

  public addCustomer(customer: NewCustomer): Observable<Customer> {
    return this.httpSvc.post<Customer>(`${this.url}`, customer);
  }

  //...
}
```

Figura 4.3: Servizio per le API riguardanti i clienti

Importante notare dalla figura che la classe estende *BaseFilterService*: questo è un servizio della libreria aziendale che fornisce funzionalità per la gestione dei filtri da applicare alle API per ottenere liste di oggetti. In quasi tutti i servizi che gestiscono chiamate API al backend è quindi stato necessario estendere questa classe.

I servizi sono stati utilizzati anche in altri casi, come ad esempio per la navigazione, che verrà approfondita successivamente.
In molti snippet di codice si può notare inoltre l'utilizzo di *ErrorsService* e di *ToastService*: questi sono presenti nella libreria fornitami dall'azienda e permettono di gestire gli errori, in genere tramite la visualizzazione di un messaggio in una finestra dialog, e di mostrare dei messaggi detti *toast*. Essi sono messaggi temporanei che vengono visualizzati in genere in un angolo dello schermo.



(a) **(b)**

Figura 4.4: Esempi di un errore (a) e di un toast (b)

4.2.2 Internationalization

L'*internationalization*, (o *i18n*) consiste nel rendere un software adattabile a diverse lingue: le traduzioni vengono separate dal resto del codice in modo da essere indipendenti e facilmente modificabili. Tramite il pacchetto *ngx-translate* è stato possibile implementare facilmente questa funzionalità, attualmente in due lingue: italiano e inglese. Ci sono infatti due file json che vanno a definire tutte le traduzioni, uno per ogni lingua. Le traduzioni vanno poi richiamate nella pagina HTML tramite l'*interpolation* di Angular.

(a) **(b)**

Figura 4.5: Esempi i18n

```
<a class="nav-link active" [ngClass]="{ 'active':activeTab==='general'}"
  (click)="changeTab('general')">
  {{'CUSTOMER.DETAIL.GENERAL' | translate}}
</a>
```

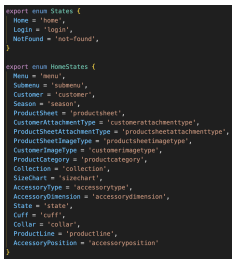
Figura 4.6: Utilizzo nell'HTML

4.2.3 Routing

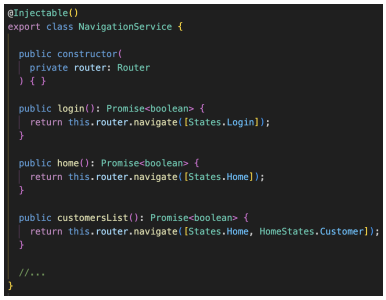
Il routing tra i componenti della wep app è realizzato tramite il routing standard di Angular. La navigazione parte dalla pagina principale che definisce tre percorsi: *login*, *not found* e *home*. I primi due descrivono semplicemente l'omonimo componente, senza nessun tipo di routing interno. Il terzo invece è il modulo principale della web app: in esso c'è un altra gestione dei reindirizzamenti che definisce tutti i percorsi ai moduli figli. A sua volta poi ogni modulo definisce il routing ai componenti che gestisce.

Per effettuare la vera e propria navigazione è stato creato un servizio di navigazione: per ogni percorso disponibile è presente un metodo che richiama il metodo *Navigate* della classe *Router* di Angular.
Per costruire l'[G] URL da utilizzare sono stati definiti degli [G] enumeratori che rappresentano i vari pezzi da unire a eventuali parametri per definire la stringa finale.

(a) home.module.ts



(b) Definizione stringhe per URL



(c) Metodi per il cambio pagina

Figura 4.7: Gestione del routing

Successivamente è presente il codice per la navigazione tramite parametri. In questo caso è visualizzato il modulo di un cliente dove vengono definiti i componenti figli, cioè la lista (che è di default se non viene specificato quale tra i due) e il dettaglio. Il dettaglio deve avere un parametro che indica l'Id del cliente da visualizzare. Questo viene specificato nel metodo del NavigationServce.

Figura 4.8: Definizione percorsi figli e parametrizzazione

Figura 4.9: Inserimento del parametro nell'URL

4.2.4 Autenticazione

L'autenticazione è anche questa gestita tramite diversi servizi dell'azienda. Sarà l'AuthService colui che si occuperà di eseguire il login, il logout e di verificare se l'utente quando accede alla web app possiede già un token di autenticazione valido. Questo servizio sfrutta in particolare il protocollo OAuth2.
In seguito si può osservare la funzione innescata dal click del bottone di login.

```
public handleUserLogin(): void {
  if (!this.formGroup.valid) {
    this.toastSvc.mandatoryInfoMissing();
    return;
  }
  const formValue: { username: string, password: string } = this.formGroup.value;
  this.isLoading = true;
  this.authSvc.login(formValue.username, formValue.password)
    .then(
      () => this.handleLoginSuccess(),
      err => {
        this.isLoading = false;
        if (err && err.status === 400) {
          this.toastSvc.localizedError('ERROR-CODES.INVALID-CREDENTIALS');
        } else {
          this.errorsSvc.handle(err);
        }
      }
    );
}
```

Figura 4.10: Funzione di autenticazione

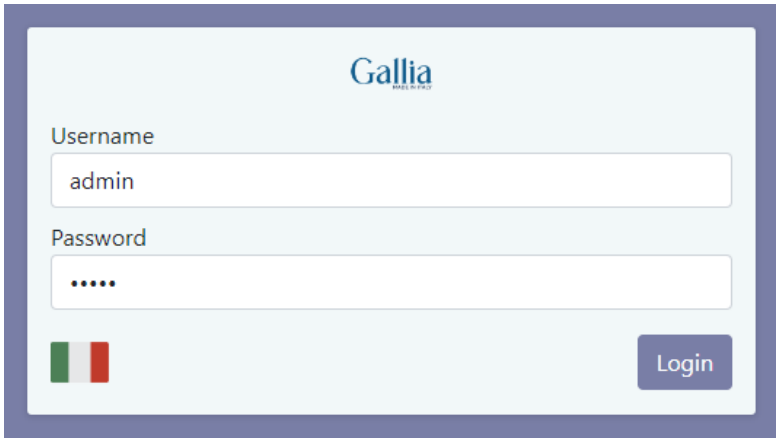


Figura 4.11: Pagina del login

4.2.5 Navbar

È stato anche realizzato un componente per la *navbar*, ossia la barra di navigazione presente in alto in tutte le pagine della web app. In modo da poterla applicare direttamente in tutti i componenti è stata inserita semplicemente all'inizio della pagina home, sopra il router che gestisce quale pagina visualizzare.

```
<lib-nav-bar></lib-nav-bar>

<div id="router-container">
  <router-outlet></router-outlet>
</div>
```

Figura 4.12: home.component.html

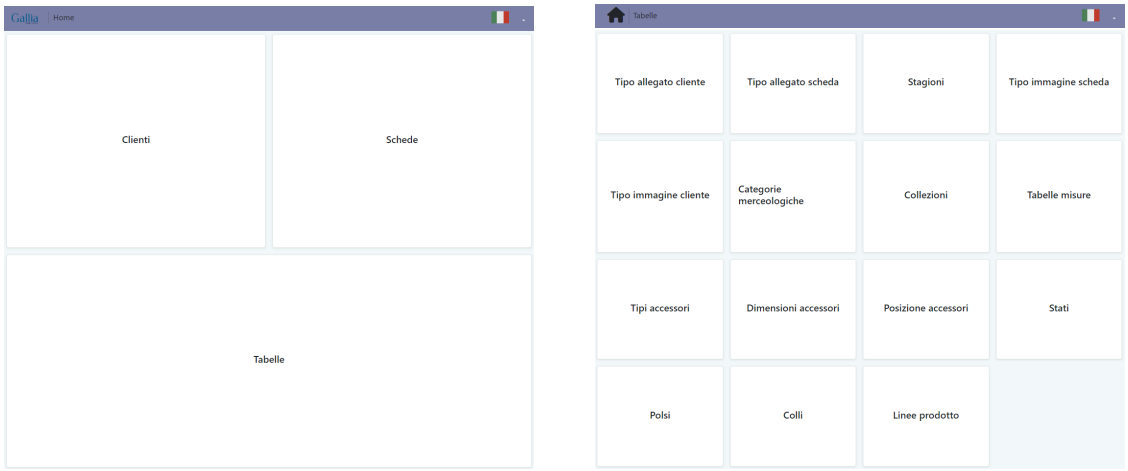
I componenti devono quindi sfruttare un servizio denominato *NavbarService* per poter comunicare con la navbar. Questo è utilizzato per poter dire al componente navbar quale icona e titolo rappresentare, oltre che per poter indicare quale sarà la funzione da eseguire quando viene cliccata l'icona, in genere per tornare alla pagina precedente.

Oltre alla possibilità di tornare indietro la navbar permette di cambiare lingua e di effettuare il logout.

Figura 4.13: Navbar

4.2.6 Menù e sottomenù

Il frontend è organizzato tramite due schermate. La prima, definita *'Menù'*, è la schermata principale: contiene i tasti per accedere alla lista dei clienti e alla lista delle schede. Da questa si può anche accedere anche all'altra schermata, definita *'Sottomenù'*: in questa ci sono tutti i bottoni per accedere a tutte le schermate della web app.
Questi componenti sono costituiti con la libreria Bootstrap da *card* organizzate tramite un *grid layout*, per poter essere responsive.



(a) Menù **(b)** Sottomenù

Figura 4.14: Navigazione principale

4.2.7 Tabelle e liste

Gran parte delle funzionalità richieste riguardavano la rappresentazione di dati in tabelle e liste. Questo quindi ha richiesto l'utilizzo di componenti specifici per questo scopo.
Per la maggior parte dei casi sono state utilizzate delle Kendo Grid, presenti nella libreria Kendo UI per Angular. Questi componenti sono spesso usati dall'azienda ospitante per questo tipo di rappresentazioni. Grande vantaggio è che forniscono la possibilità di paginare molto facilmente i dati ottenuti.

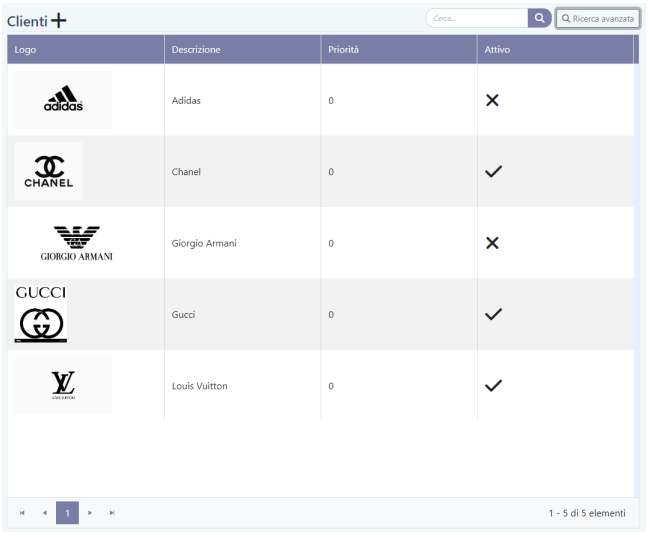


Figura 4.15: Esempio di una kendo grid

Non è nemmeno necessario ottenere i dati dal file TypeScript, visto che grazie a un decoratore denominato *sogeaGridFilterDataBinding* creato dall'azienda i dati vengono automaticamente caricati nella tabella.
Sono stati usati due modi per riempire le colonne. Il primo sfrutta l'attributo *field* delle *kendo-column*, che se corrisponde al nome di un attributo dell'oggetto definito nel servizio, va a prenderne il valore e rappresentarlo nella cella. Il secondo è utile nel caso debbano essere definiti elementi complessi, visto che nel primo si rappresenterebbe solo il dato testuale. Si definisce quindi un *ng-template* con la direttiva *kendoGridCellTemplate* e l'attributo *let-dataItem* con al suo interno l'HTML da visualizzare: qui si potrà quindi accedervi tramite la variabile *dataItem*.

```
<kendo-grid [pageable]="true" [resizable]="true" [sogeaGridFilterDataBinding]="customerSvc" class="h-100"
  (cellClick)="openDetail($event)">
  <kendo-grid-column [title]="'PRODUCT-SHEETS.CUSTOMER-LIST.LOGO' | translate">
    <ng-template kendoGridCellTemplate let-dataItem>
      <img class="logo-cliente" *ngIf="dataItem?.logoFormat" [src]="logoUrl(dataItem)" height="80" width="80" />
      <div *ngIf="!dataItem?.logoFormat">
        <i class="fa-solid fa-image-slash fa-4x"></i>
      </div>
    </ng-template>
  </kendo-grid-column>
  <kendo-grid-column field="description" [title]="'GENERAL.DESCRIPTION' | translate">
  </kendo-grid-column>
  <kendo-grid-column field="priority" [title]="'CUSTOMER.PRIORITY'|translate">
  </kendo-grid-column>
  <kendo-grid-checkbox-column field="enabled" [title]="'GENERAL.ENABLED' | translate">
    <ng-template kendoGridCellTemplate let-dataItem>
      <span *ngIf="dataItem.enabled" (click)="disabilita($event, dataItem)">
        <i class="fas fa-check fa-2xl"></i>
      </span>
      <span *ngIf="!dataItem.enabled" (click)="abilita($event, dataItem)">
        <i class="fas fa-times fa-2xl"></i>
      </span>
    </ng-template>
  </kendo-grid-checkbox-column>
</kendo-grid>
```

Figura 4.16: HTML di una kendo grid

Nella maggior parte dei casi questo componente è stato utilizzato in questo modo. In alcuni casi è stato necessario però modificarne l'utilizzo. Ad esempio per la rappresentazione delle liste di allegati è stata utilizzata una singola colonna. Anche qui è stato utilizzato un *ng-template* per poter creare degli elementi in modo da poterli personalizzare al massimo. Grazie a questo trucco era possibile una visualizzazione più elegante per la lista.

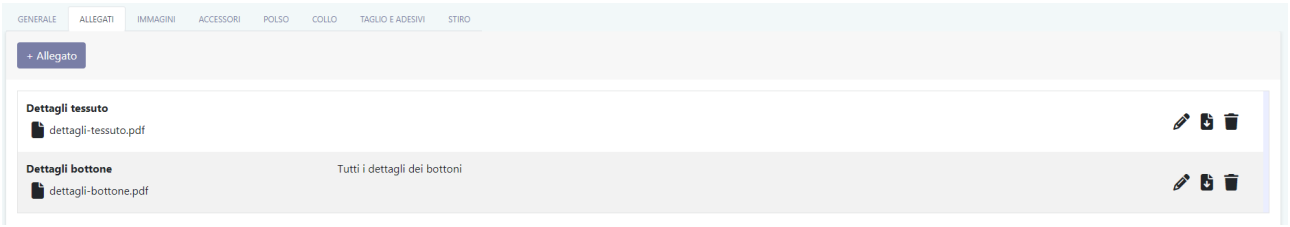


Figura 4.17: Grafica con una singola colonna

```
<kendo-grid [autoSize]="true" [hideHeader]="true" [data]="attachments">
  <kendo-grid-column>
    <ng-template kendoGridCellTemplate let-dataItem>
      <div class="row">
        <div [ngClass]="typeColumn">
          <div class="d-flex flex-column">
            <div class="fw-bold">
              {{dataItem.attachmentTypeDescription}}
            </div>
            <div class="d-flex flex-row align-items-center">
              <span class="p-2">
                <i class="fa-solid fa-file fa-xl"></i>
              </span>
              <div class="text-truncate">
                {{dataItem.fileName}}.{{dataItem.fileExtension}}
              </div>
            </div>
          </div>
        </div>
        <div [ngClass]="descriptionColumn" class="truncate-3-lines">
          {{dataItem.description}}
        </div>
        <div [ngClass]="buttonsColumn" class="btn-group align-items-center">
          <span class="p-2" (click)="edit(dataItem)">
            <i class="fa-solid fa-pencil fa-xl"></i>
          </span>
          <span class="p-2" (click)="save(dataItem)">
            <i class="fa-solid fa-file-arrow-down fa-xl"></i>
          </span>
          <span class="p-2" (click)="delete(dataItem)">
            <i class="fa-solid fa-trash fa-xl"></i>
          </span>
        </div>
      </div>
    </ng-template>
  </kendo-grid-column>
</kendo-grid>
```

Figura 4.18: Rappresentazione a singola colonna

Assieme a queste tabelle è stato spesso utilizzato un altro componente sviluppato dalla libreria aziendale, visibile nella figura 4.15. Chiamato *sogea-advanced-search-bar*, consiste in una barra per la ricerca semplice tramite keyword e un bottone per la ricerca avanzata. La ricerca avanzata consiste in alcuni filtri da applicare alla lista, definiti tramite un form.

4.2.8 Dettaglio

La maggior parte delle schermate del sistema sono composte da una singola tabella, e sono tutte molto simili tra loro. Ci sono però alcuni casi in cui la kendo grid non è bastata per rappresentare tutti i dati di alcuni oggetti. In questi casi si è creato un sottocomponente che rappresentasse un singolo oggetto nella lista. Questo componente viene aperto tramite un click su una riga della tabella, che richiama il servizio di navigazione.

```
public openDetail(event: CellClickEvent): void {
  this.navigationSvc.customerDetail(event.dataItem.id);
}
```

Figura 4.19: Funzione scatenata al click su una riga

Ogni dettaglio è suddiviso da dei tab di Bootstrap: in questo modo si hanno più schede per poter organizzare meglio gli attributi definiti negli oggetti del dominio. È sempre presente il tab *Generale*, che contiene tutti gli attributi più semplici e importanti dell'oggetto.
Nella figura seguente è presente il tab generale del dettaglio di un collo.

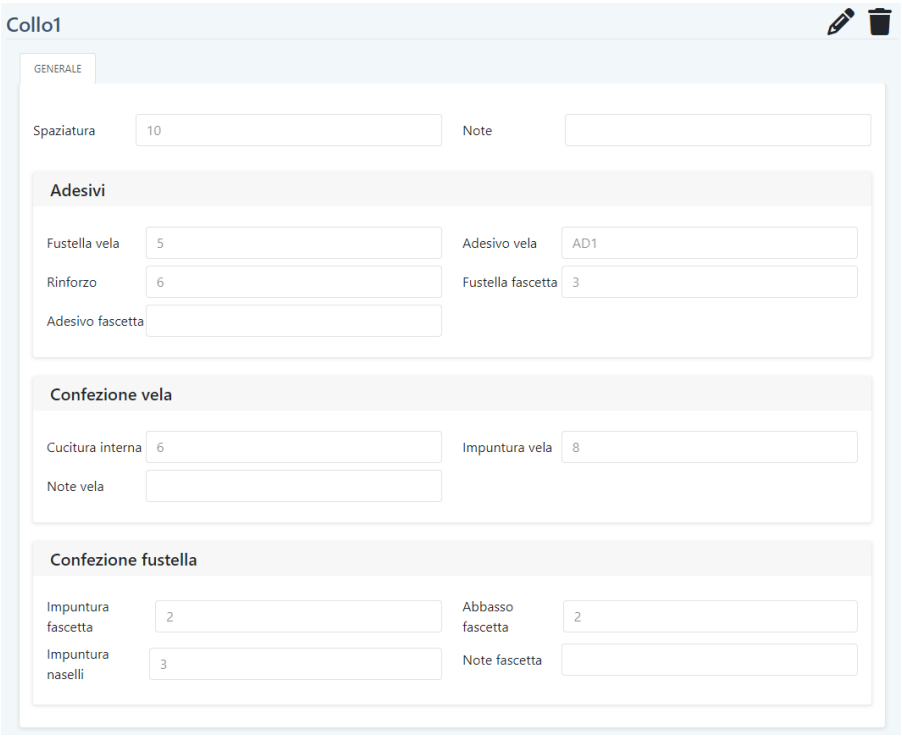


Figura 4.20: Tab generale di un collo

In particolare la parte più complessa del sistema sono i dettagli dei clienti e delle schede. Il dettaglio del cliente è composto da quattro tab: *Generale*, *Note*, *Allegati* e *Immagini*.

- **Generale**: come detto prima, sono presenti attualmente pochi attributi (attivo, priorità, linea prodotto), ma in futuro potrebbero essere aggiunti altri campi, quindi nonostante sia scarno questo viene lo stesso mantenuto.
- **Note**: tramite una list-group di Bootstrap è stata creata una lista di note, con le funzionalità per poterle aggiungere, modificare, eliminare e spostare tramite delle frecce.

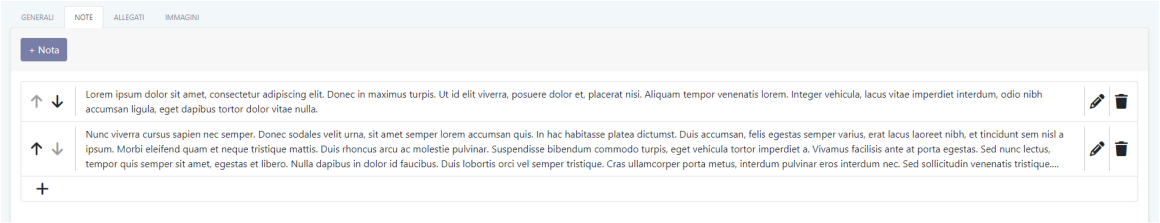


Figura 4.21: Lista delle note

- **Allegati**: questo è il tab in cui è stato sfruttato il trucco della kendo grid a singola colonna.
- **Immagini**: in questo tab invece si è creato un componente apposito per la visualizzazione della singola immagine. Si è poi utilizzato un **ngFor* per poter visualizzare tutte le immagini della lista. Anche in questo caso è stato utilizzato il grid layout di Bootstrap per poter avere una pagina responsive.

```
<!-- IMAGES -->
<div class="tab-pane h-100" [ngClass]="{ 'active':activeTab==='images'}">
  <div class="card h-100">
    <div class="card-header p-3">
      <button type="button" class="btn btn-primary" (click)="addImage()">+ Immagine</button>
    </div>
    <div class="card-body overflow-auto">
      <div class="row row-cols-1 row-cols-sm-1 row-cols-md-2 row-cols-lg-3 row-cols-xl-4 row-cols-xxl-5 g-2">
        <lib-image-list-item (downloadEvent)="saveImage($event)" (editEvent)="editImageDescription($event)"
          (deleteEvent)="deleteImage($event)" class="mt-1 mb-2 col" *ngFor="let image of customer?.images"
          [image]="image">
        </lib-image-list-item>
      </div>
      <div *ngIf="!customer?.images.length">
        {{'CUSTOMER.DETAIL.NO-IMAGES' | translate}}
      </div>
    </div>
  </div>
</div>
```

Figura 4.22: Tab dell'immagine

```
<div class="card h-100">
  <img [src]='logo' class="card-img-top" alt="..." />
  <div class="card-body">
    <h5 class="card-title">{{image.imageTypeDescription}}</h5>
    <div class="d-flex flex-row justify-content-between">
      <div>
        {{image.description}}
      </div>
      <div>
        <span class="p-2" (click)="downloadImage()">
          <i class="fa-solid fa-arrow-down"></i>
        </span>
        <span class="p-2" (click)="editImageDescription()">
          <i class="fa-solid fa-pencil"></i>
        </span>
        <span class="p-2" (click)="deleteImage()">
          <i class="fa-solid fa-trash"></i>
        </span>
      </div>
    </div>
  </div>
</div>
```

Figura 4.23: image-list-item.component.html

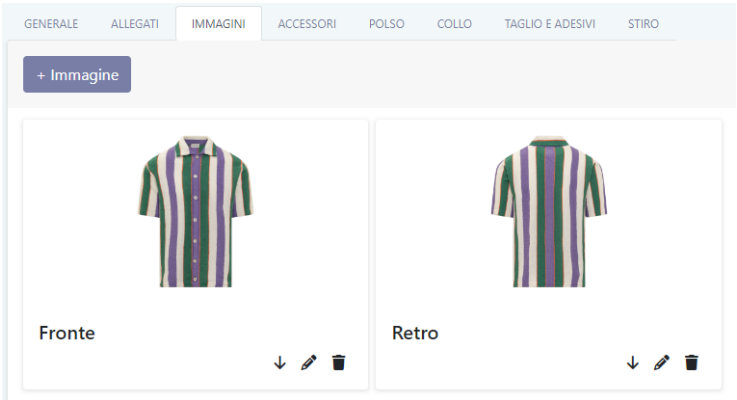


Figura 4.24: Grafica tab immagini

Il dettaglio della scheda invece è composto da otto tab:

- **Generale**: in questo caso è abbastanza complesso in quanto contiene diversi attributi, modificabili tramite un form.

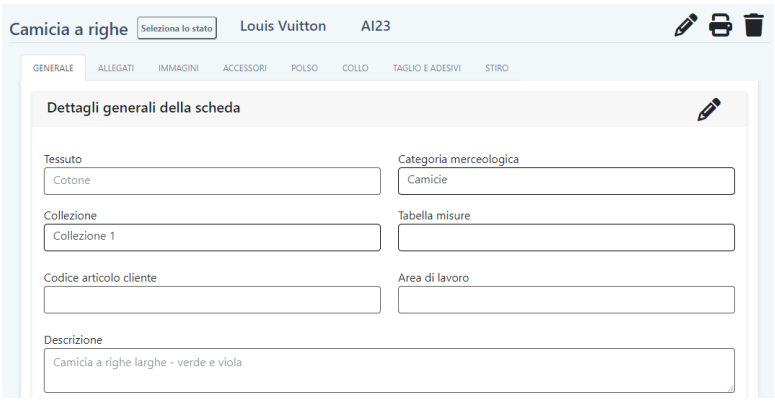


Figura 4.25: Tab generale della scheda

- **Allegati**: il componente è lo stesso utilizzato per il dettaglio del cliente.
- **Immagini**: anche qui il componente è lo stesso utilizzato per il dettaglio del cliente.
- **Accessori**: qui è presente una kendo grid che contiene la lista degli accessori; anche in questo caso c'è la possibilità di modificare l'ordine tramite delle frecce.

Figura 4.26: Grafica tab accessori

- **Polso**: in questo tab è presente una *dropdownlist* di Kendo, che permette di selezionare un polso e visualizzare tutti i relativi attributi.

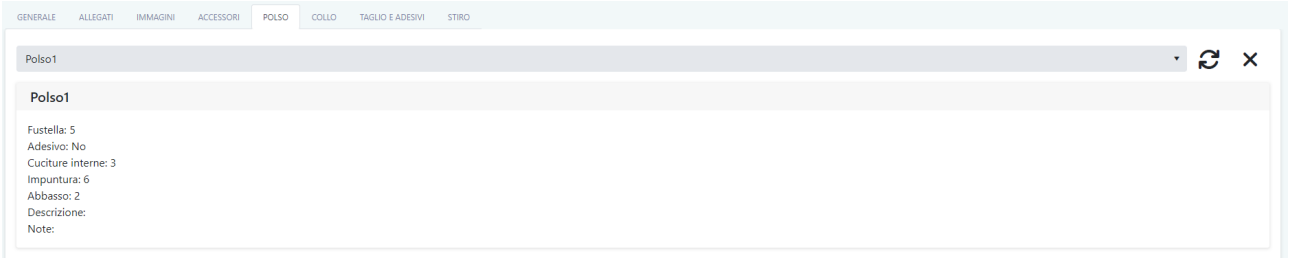


Figura 4.27: Grafica tab polso

- **Collo**: è come il tab precedente, ma contiene il collo relativo alla scheda.

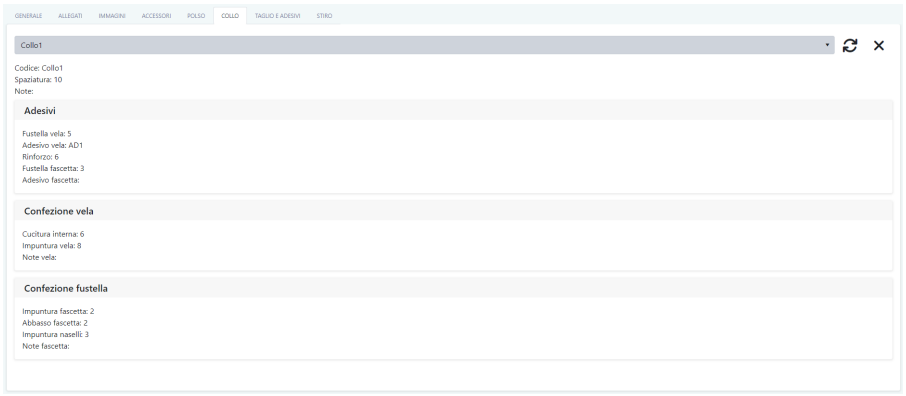


Figura 4.28: Grafica tab collo

- **Taglio e adesivi**: contiene solo due textarea che contengono le note di taglio e le nota per gli adesivi.
- **Stiro**: contiene una singola textarea per le note di stiro.

4.2.9 Modali

Per la comunicazione con l'utente sono stati spesso utilizzati dei dialog. Grazie alla libreria aziendale e ai dialog di Bootstrap è stato molto semplice creare e utilizzare questi componenti. Quasi sempre contengono un form reattivo gestito quasi interamente da Angular per poter inserire i dati richiesti e due bottoni per confermare o annullare l'operazione. Ad esempio sotto viene riportato l'HTML di un dialog per la creazione di un nuovo stato. Particolarità di questo dialog è che è stato utilizzato un *color-picker*, ossia un componente che permette di selezionare un colore tramite una palette: esso è stato preso dal pacchetto *ngx-color-picker*.

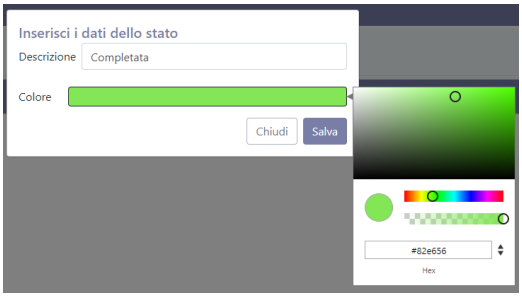


Figura 4.29: Dialog per l'inserimento di uno stato

```
<div class="modal-content">
  <div class="modal-header pb-0">
    <h5 class="modal-title">{{'STATE.NEW'|translate}}</h5>
  </div>
  <div class="modal-body py-0">
    <form class="needs-submit" [formGroup]="formGroup">
      <div class="d-flex flex-row align-items-center">
        <label for="descriptionInput" class="pe-2">{{'GENERAL.DESCRIPTION'|translate}}</label>
        <input kendoTextBox id="descriptionInput" class="form-control rounded" formControlName="description">
      </div>
      <div class="d-flex flex-row align-items-center pt-4">
        <label for="colorInput" class="pe-4">{{'STATE.COLOR'|translate}}</label>
        <input class="w-100 border border-dark rounded" type="button" id="colorInput" [style.background]="color" [(colorPicker)]="color"
        [cpOKButton]="false" />
      </div>
    </form>
  </div>
  <div class="modal-footer">
    <button type="button" class="btn btn-secondary" (click)="cancel()">{{'GENERAL.CLOSE'|translate}}</button>
    <button type="button" class="btn btn-primary" (click)="save()"
      [disabled]="!formGroup.valid">{{'GENERAL.SAVE'|translate}}</button>
  </div>
</div>
```

Figura 4.30: Esempio di dialog

4.3 Backend

4.3.1 Dominio

Ogni aggregate root ha una cartella nel progetto denominato dominio, con all'interno i relativi file C# che definiscono le entità e i value object. La cartella prende il nome da quello dell'aggregate root, che ha anche un file omonimo che descrive l'entità e un altro che descrive l'interfaccia del repository (secondo il repository pattern).
Tutte le classi relative all'aggregate sfruttano, grazie all'ereditarietà dell'[G] Object Oriented Programming, diverse classi e interfacce già presenti nella libreria aziendale. Visto l'utilizzo ampio del DDD nei progetti dell'azienda infatti sono stati definiti gli oggetti per modellare entità, value object e il resto dei componenti necessari.

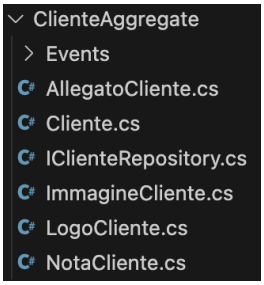


Figura 4.31: Esempio di aggregate root

- **Entità**: grazie alla classe *BaseEntity* e all'interfaccia *ITrackedEntity* vengono definiti gli attributi comuni a tutte le entità, in particolare un Id univoco di tipo *Guid* e le informazioni per il tracciamento di creazione e modifica dell'oggetto. Sono poi definiti nella classe tutti gli attributi e i metodi da cui è composta l'entità.
- **Aggregate Root**: oltre a essere un entità implementa anche le interfacce *IAggregateRoot*, *IOwnedByTenant* e *IConcurrencyEntity*. La prima fornisce la gestione per gli invarianti di un aggregate root (se necessario si dovrà effettuare l'[G] override); la seconda è utilizzata per la gestione dei tenant, non trattati in questo caso ma lo stesso utilizzata per eventuali sviluppi futuri; la terza infine gestisce la concorrenza nel caso di più modifiche contemporanee allo stesso oggetto. In questa classe è anche presente la gestione degli eventi: questi sono utilizzati per tenere uno storico delle modifiche effettuate all'oggetto in caso di necessità o di perdita di dati. Per ogni metodo dell'aggregate root viene definito un evento (in quanto sono l'unico modo in cui c'è comunicazione con l'esterno) che con un metodo di BaseEntity viene aggiunta a un database apposito per gli eventi. C'è quindi anche una cartella *Events* che contiene le classi degli eventi.

```
public class Cliente : BaseEntity, IAggregateRoot, ITrackedEntity, IOwnedByTenant, IConcurrencyEntity
{
  protected readonly List<NotaCliente> note;
  protected readonly List<AllegatoCliente> allegati;
  protected readonly List<ImmagineCliente> immagini;

  public Cliente()
  { }

  public Cliente(string descrizione)
  {
    this.Id = Guid.NewGuid();
    this.Descrizione = descrizione;
    this.Abilitato = true;
    this.Priorita = 0;
    this.AddDomainEvent(new ClienteCreatoEvento(descrizione));
  }
```

Figura 4.32: Esempio di codice di un aggregate root

```
internal class ClientePrioritaAggiornataEvento : DomainEvent, IEntityDomainEvent
{
  public ClientePrioritaAggiornataEvento(int priorita)
  {
    Priorita = priorita;
  }

  public int Priorita { get; private set; }

  public Guid EntityId { get; set; }
}
```

(a) Esempio di un evento

```
public void AggiornaPriorita(int priorita)
{
  this.Priorita = priorita;
  this.AddDomainEvent(new ClientePrioritaAggiornataEvento(priorita));
}
```

(b) Esempio dell'aggiunta di un evento

Figura 4.33: Gestione delgli eventi

- **Value Object**: questi ereditano solo dalla classe *Value Object* diversi attributi e metodi per sapere quando è stato creato, se è uguale a un altro oggetto, per clonarlo e vari altri.

```
public class LogoCliente : ValueObject
{
  public LogoCliente(byte[] logo, string formato)
  {
    this.Logo = logo;
    this.Formato = formato;
  }
```

Figura 4.34: Esempio di codice di un value object

- **Repository**: sono interfacce che definiscono quali metodi dovranno essere implementati per la gestione di una repository di un entità. Sono quasi sempre *AddAsync*, *FindAsync* e *DeleteAsync* che rispettivamento aggiungono, trovano ed eliminano un oggetto in modo asincrono.

```
public interface IClienteRepository
{
  Task AddAsync(Cliente entity);

  Task<Cliente> FindAsync(Guid id);

  Task DeleteAsync(Cliente entity);
}
```

Figura 4.35: Esempio di interfaccia repository

4.3.2 DTO

In questo progetto sono presenti, sempre divisi per aggregate root, i DTO utilizzati dal sistema. Ce ne sono di diversi tipi, utilizzati per scopi diversi:

- fornire delle indicazioni riguardanti la chiamata delle API, infatti vengono definiti alcuni DTO per descrivere quali saranno i campi del payload di ogni richiesta, per renderla più chiara;

```
public class CreateCustomerLogo : BaseDto
{
  public byte[] Logo { get; set; }

  public string LogoFormat { get; set; }

  public byte[] Timestamp { get; set; }
}
```

Figura 4.36: Esempio DTO utilizzato per un payload

  tra questi ci sono tra l'altro i DTO utilizzati per ottenere le liste di oggetti: sono tutte classi figlie di *BasePaginatedDTO*, che fornisce i campi per la paginazione; vanno poi definiti in ogni oggetto gli eventuali campi per filtrare le liste;

```cs
public class BasePaginatedDTO
{
  public int Page { get; set; }

  public int PageSize { get; set; }

  public string Search { get; set; }
}
```

(a) BasePaginatedDTO.cs

```cs
public class GetListCustomer : BasePaginatedDTO
{
  public bool Enabled { get; set; }

  public bool Disabled { get; set; }
}
```

(b) Utilizzo di BasePaginatedDTO ed esempio di filtri

Figura 4.37: Esempio di un filtro

- definire degli oggetti che saranno nella risposta delle API, che possono essere diversi da quelli rappresentati nel dominio;

```cs
public class CustomerListItem : BaseDto
{
  public Guid Id { get; set; }

  public string Description { get; set; }

  public bool Enabled { get; set; }

  public int Priority { get; set; }

  public byte[] Logo { get; set; }

  public string LogoFormat { get; set; }

  public byte[] Timestamp { get; set; }
}
```

Figura 4.38: Esempio DTO utilizzato per il body di una risposta

4.3.3 Command

Qui ci sono le classi che definiscono, come descritto nel capitolo 3, i comandi e i rispettivi handler per le azioni che eseguono operazioni di scrittura nel database. I comandi sono classi molto semplici: contengono solo gli attributi che verranno utilizzati per effettuare l'operazione.

```cs
public class AddLogoCommand : IDomainCommand
{
  public AddLogoCommand(Guid id, byte[] logo, string format, byte[] timestamp)
  {
    this.Id = id;
    this.Logo = logo;
    this.Format = format;
    this.Timestamp = timestamp;
  }

  public Guid Id { get; private set; }

  public byte[] Logo { get; private set; }

  public string Format { get; private set; }

  public byte[] Timestamp { get; private set; }
}
```

Figura 4.39: Esempio di un command

Tramite l'utilizzo della libreria aziendale, gli handler invece sono coloro che eseguono l'operazione. Ognuno infatti contiene un metodo *Handle* che va a gestire la richiesta definita dalla classe command. Sarà un'entità denominata *DomainBus* che all'inoltro di un comando andrà a cercare l'handler che gestisce esattamente quella classe.

```
[DecorateDomainBusHandler(typeof(TransactionalDecorator<AddLogoCommand>))]
internal class AddLogoCommandHandler : IDomainCommandHandler<AddLogoCommand>
{
  public AddLogoCommandHandler(IClienteRepository repository)
  {
    this.Repository = repository;
  }

  private IClienteRepository Repository { get; }

  public async Task<Unit> Handle(AddLogoCommand request, CancellationToken cancellationToken)
  {
    Cliente customer = await this.Repository.FindAsync(request.Id);
    customer.AggiornaLogo(request.Logo, request.Format);
    customer.SetModifiedWithOriginalTimestamp(request.Timestamp);
    return this.Completed();
  }
}
```

Figura 4.40: Esempio di un command handler

4.3.4 Query

La logica del progetto query è simile a quella del command. L'unica differenza è che qui ci sono solo le classi che definiscono le query, ma non gli handler, che sono in un altro progetto definito successivamente. Inoltre, implementando anche la [G]classe generica *IQuery*, devono dichiarare come parametro di tipo il DTO che definisce il risultato della query.
La maggior parte delle query sono sempre le stesse, infatti richiedono solo di ottenere un oggetto tramite un Id oppure di ottenere una lista di oggetti secondo dei filtri. Sono state create quindi diverse classi generiche che definiscono le query più comuni, in modo da poterle riutilizzare per quasi tutti gli aggregate root.

```
public class GenericByIdQuery<T> : IQuery<T>
{
  public GenericByIdQuery(Guid id)
  {
    this.Id = id;
  }

  public Guid Id { get; }
}
```

(a) Query per la lettura di un singolo elemento

```
public class GenericListItemQuery<T> : IQuery<BaseListDto<T>>, IPaginatedQuery
{
  public GenericListItemQuery(int page, int pageSize, string search)
  {
    this.Page = page;
    this.PageSize = pageSize;
    this.Search = search;
  }

  public int Page { get; private set; }

  public int PageSize { get; private set; }

  public string Search { get; private set; }
}
```

(b) Query per la lettura di una lista di elementi

Figura 4.41: Query generali

In alcuni casi è stato però necessario creare delle query specifiche, come nella foto successiva. Tra l'altro non sempre si nota la differenza tra una query generica e specifica da questa classe, perchè ci sono differenze nell'handler che le gestisce.

```
public class CustomerListItemQuery : IQuery<BaseListDto<CustomerListItem>>, IPaginatedQuery
{
  public CustomerListItemQuery(int page, int pageSize, string search, bool enabled, bool disabled)
  {
    this.Page = page;
    this.PageSize = pageSize;
    this.Search = search;
    this.Enabled = enabled;
    this.Disabled = disabled;
  }

  public int Page { get; private set; }

  public int PageSize { get; private set; }

  public string Search { get; private set; }

  public bool Enabled { get; private set; }

  public bool Disabled { get; private set; }
}
```

Figura 4.42: Esempio di una query specifica

4.3.5 ApplicationService

Qui sono stati sviluppati gli handler per gestire le API. Seguendo una struttura definita dall'azienda e utilizzando la libreria ogni richiesta HTTP inviata al microservizio arriva in un controller definito dal nome messo come decoratore come nella foto successiva. Ogni controller inoltre ha due attributi, che sono gli oggetti responsabili di eseguire i comandi e le query.

```
[Authorize(AuthorizeAttribute.AccessLevel.Public)]
[Handler("Customers")]
public class CustomersHandler : IHandler
{
  public CustomersHandler(IQueryProcessor queryProcessor, ISyncStopOnExceptionDomainBus domainBus)
  {
    this.QueryProcessor = queryProcessor;
    this.DomainBus = domainBus;
  }

  private ISyncStopOnExceptionDomainBus DomainBus { get; }

  private IQueryProcessor QueryProcessor { get; }
```

Figura 4.43: Esempio di handler

Per ogni API necessaria viene poi definito un metodo: questo avrà un decoratore per specificare il tipo della richiesta HTTP e come dev'esserne l'URL. I parametri dei metodi, grazie ad altri decoratori, andranno a definire quali saranno i parametri e il payload della richiesta.

```
[ActionHandler(ActionHandlerMeanings.Update, ActionHandlerNaming.DerivedFromMeaning)]
public async Task<Customer> UpdateDescription([UseToAccessTo("description")] Guid id, [Payload] UpdateDescriptionCustomer payload)
{
  await this.DomainBus.Send(new UpdateDescriptionCommand(id, payload.Description, payload.Timestamp));
  return await this.QueryProcessor.Process(new GenericByIdQuery<Customer>(id));
}
```

Figura 4.44: Esempio di API

4.3.6 Infrastructure

4.3.6.1 Persistence

Come descritto nella capitolo 3, questa cartella contiene la gestione della scrittura nel database ed è composta da due progetti: uno per ogni database e uno che si interfaccia con quello utilizzato in questo specifico caso, ossia PostgreSQL.
Aspetto fondamentale del primo progetto è quello di occuparsi delle migrations: all'avvio del sistema infatti sarà una classe definita qui a interfacciarsi con EFCore e controllare se ce ne sono da applicare.
Ci sono poi le implementazioni delle interfacce dei repository di ogni aggregate root: grazie alla classe della libreria aziendale *BaseAuthorizedRepository*, che sarà quella a eseguire le vere e proprie operazioni, sono a disposizione tutti i metodi più utilizzati per la loro gestione.

```
internal class ClienteRepository : IClienteRepository
{
  public ClienteRepository(BaseAuthorizedRepository<Cliente> baseRepository)
  {
    this.BaseRepository = baseRepository;
  }

  private BaseAuthorizedRepository<Cliente> BaseRepository { get; }

  public Task AddAsync(Cliente entity)
    => this.BaseRepository.InsertAsync(entity);

  public Task DeleteAsync(Cliente entity)
    => this.BaseRepository.DeleteAsync(entity);

  public Task<Cliente> FindAsync(Guid id)
    => this.BaseRepository.GetAsync(id);
}
```

Figura 4.45: Esempio di repository

Sono poi presenti le configurazioni degli aggregate root, per la generazione delle migrations. Qui vanno deinite le relazioni tra le tabelle, quindi chiave primaria, eventuali chiavi esterne e relazioni con altre tabelle.

```
public void Configure(EntityTypeBuilder<Cliente> builder)
{
  this.ConfigureEntity(builder).HasKey(x => x.Id);
  builder.Property(x => x.Id).ValueGeneratedNever();
  builder.OwnsMany(x => x.Allegati, val =>
  {
    this.ConfigureEntity(val).HasKey(x => x.Id);
    val.Property(x => x.Id).ValueGeneratedNever();
    val.WithOwner().HasForeignKey("ClienteId");
    val.HasOne<TipoAllegato>().WithMany().HasForeignKey(x => x.TipoAllegatoId);
  });
```

Figura 4.46: Esempio parziale di configurazione

Nell'altro progetto invece ci sono i file delle migrations generati da EFCore, che definiscono tutte le tabelle del database. Viene anche definito quale database verrà utilizzato ed eventuali altre configurazioni.

4.3.6.2 Query

Anche qui c'è la distinzione tra il progetto generale e il progetto specifico per PostgreSQL. Nel primo progetto c'è il modello di lettura dal database: ci sono tutte le classi che definiscono gli oggetti letti dal database. È simile al dominio, solo che vanno definiti unicamente gli attributi. Grazie a EFCore basta definirli con lo stesso nome delle colonne del database; inoltre per le chiavi esterne vengono definiti due attributi, uno per l'Id (che sarebbe la chiave esterna nella tabella di riferimento) e uno per l'oggetto vero e proprio, che viene ottenuto tramite le [G] *navigation property*.

```
public class Cliente : IBaseEntityModel
{
  public Guid Id { get; set; }

  public string Descrizione { get; set; }

  public bool Abilitato { get; set; }

  public int Priorita { get; set; }

  public string Logo_Formato { get; set; }

  public byte[] Logo_Logo { get; set; }

  public Guid? LineaProdottoId { get; set; }

  public LineaProdotto? LineaProdotto { get; set; }

  public List<NotaCliente> Note { get; set; }

  public List<AllegatoCliente> Allegati { get; set; }

  public List<ImmagineCliente> Immagini { get; set; }

  public byte[] Timestamp { get; set; }
}
```

Figura 4.47: Esempio del modello di cliente

Questi modelli per la lettura vanno poi mappati nei DTO: per facilitare questo compito è stata utilizzata la libreria Automapper. Questa libreria consente di trasferire i dati da un oggetto all'altro in modo veloce ed automatico. Per ogni aggregate root c'è quindi un file che definisce le configurazioni per il mapping tra i vari oggetti, che possono essere più di uno in caso di diverse rappresentazioni dello stesso oggetto.

```
public IMapperConfigurationExpression AddMappings(IMapperConfigurationExpression cfg)
{
  cfg.CreateMap<Cliente, CustomerListItem>()
    .ForMember(x => x.Id, opt => opt.MapFrom(x => x.Id))
    .ForMember(x => x.Description, opt => opt.MapFrom(x => x.Descrizione))
    .ForMember(x => x.Enabled, opt => opt.MapFrom(x => x.Abilitato))
    .ForMember(x => x.Priority, opt => opt.MapFrom(x => x.Priorita))
    .ForMember(x => x.Logo, opt => opt.MapFrom(x => x.Logo_Logo))
    .ForMember(x => x.LogoFormat, opt => opt.MapFrom(x => x.Logo_Formato))
    .ForMember(x => x.Timestamp, opt => opt.MapFrom(x => x.Timestamp))
    ;
```

Figura 4.48: Esempio dell'utilizzo di Automapper

Infine ci sono gli handler delle query: come detto nella sezione 4.3.4, sono state generalizzate visto che sono molto simili tra loro. Ci sono quindi delle classi generiche che permettono di ottenere un singolo oggetto o una lista di oggetti: per poi filtrare i risultati in base al risultato richiesto viene utilizzata la tecnologia LINQ.

```
public async Task<X> Handle(GenericByIdQuery<X> query, CancellationToken cancellationToken)
{
  return await Context.Set<T>()
    .Where(x => x.Id == query.Id)
    .ProjectTo<X>(Mapper.ConfigurationProvider)
    .FirstOrDefaultAsync();
}
```

Figura 4.49: Esempio della gestione di una query

Ci sono poi anche le query specifiche, che come nell'esempio seguente richiedono attenzioni più particolari: in questo caso è rappresentata la gestione dei filtri per la lista delle schede.

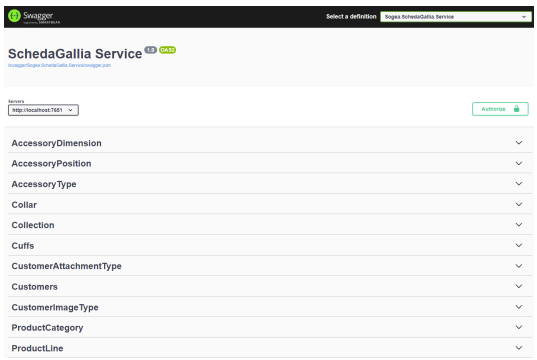
```
public async Task<BaseListDto<ProductSheetListItem>> Handle(ProductSheetListItemQuery query, CancellationToken ct)
{
  IQueryable<Scheda> lista = Context.Set<Scheda>();

  if (!string.IsNullOrEmpty(query.Search)) lista = lista.Where(x => x.Modello.ToLower().Contains(query.Search.ToLower()));
  if (isGuidValid(query.CustomerId)) lista = lista.Where(x => x.ClienteId == query.CustomerId);
  if (query.SeasonIdList != null && query.SeasonIdList.Count() > 0)
    lista = lista.Where(x => query.SeasonIdList.Contains(x.StagioneId));
  if (query.ProductCategoryIdList != null && query.ProductCategoryIdList.Count() > 0)
    lista = lista.Where(x => query.ProductCategoryIdList.Contains(x.CategoriaMerceologicaId));
  if (query.CollectionIdList != null && query.CollectionIdList.Count() > 0)
    lista = lista.Where(x => query.CollectionIdList.Contains(x.CollezioneId));
  if (query.SizeChartIdList != null && query.SizeChartIdList.Count() > 0)
    lista = lista.Where(x => query.SizeChartIdList.Contains(x.TabellaMisureId));
  if (query.StateIdList != null && query.StateIdList.Count() > 0)
    lista = lista.Where(x => query.StateIdList.Contains(x.StatoId));

  return new BaseListDto<ProductSheetListItem>()
  {
    Count = await lista.CountAsync(),
    Values = await lista.ApplyPagination(query).ProjectTo<ProductSheetListItem>(Mapper.ConfigurationProvider).ToListAsync(ct)
  };
}
```

Figura 4.50: Query handler per la lista delle schede

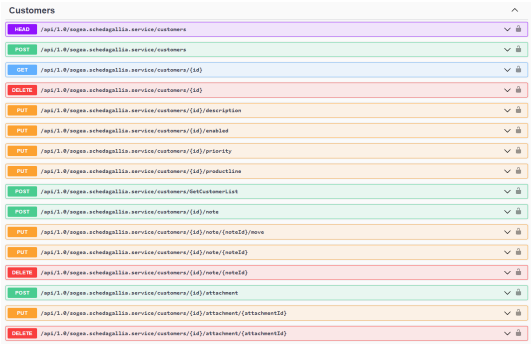
Nel secondo progetto c'è invece la configurazione dei tipi apposita per PostgreSQL.

4.3.7 Swagger

Il testing delle API è stato facilitato grazie all'utilizzo di Swagger. Questo infatti è integrato nella libreria aziendale e permette di visualizzare e testare manualmente tutte le API fornite dal microservizio. Grazie ad esso, come si nota dalle foto seguenti, viene anche autogenerata la documentazione delle API.



(a) Lista degli aggregate root



(b) Lista delle API di un singolo aggregate

Figura 4.51: Swagger

4.3.8 Jasper

È stata creata una API per la generazione dei report: in particolare è stato creato un servizio che, dato un Id di una scheda, effettua una query per ottenere l'oggetto e tramite una classe detta *CreaReportService* ritorna il file PDF. Il metodo *CreaReportAsync* utilizza in particolare la libreria *RestSharp* per effettuare una richiesta HTTP al server Jasper, dove sono definiti i template dei report.

```
[ActionHandler(ActionHandlerMeanings.Procedure, ActionHandlerNaming.DerivedFromMeaning)]
public async Task<IStreamData> CreateJasperSheet([UseToAccessTo("report")] Guid id, [Payload] PrintReport payload)
{
  ProductSheet scheda = await QueryProcessor.Process(new GenericByIdQuery<ProductSheet>(id));
  string json = JsonConvert.SerializeObject(new List<ProductSheet> { scheda });
  return await Service.CreaReportAsync(json, payload.Resource);
}
```

Figura 4.52: API per la creazione del report

4.4 Deploy

In ognuna delle due repository è presente una cartella con dei file batch per la creazione e la pubblicazione dell'immagine docker. I branch di test contengono in particolare le impostazioni per il deploy su un ambiente di testing, che nel caso di questo progetto era una macchina della rete locale dell'azienda. I branch di release invece contengono le impostazioni (in particolare gli indirizzi IP e le porte) per il deploy nell'ambiente del cliente: tramite un tunnel SSH viene effettuato il collegamento con la macchina del cliente e tramite Portainer viene creato e fatto partire lo stack con tutti i container necessari.

Capitolo 5

Conclusioni

5.1 Valutazione strumenti utilizzati

Molti strumenti erano in realtà già di mia conoscenza, quindi non ho avuto particolari difficoltà nell'utilizzarli. In particolare VSCode, pgAdmin e l'utilizzo di TypeScript e HTML erano tecnologie che mi era capitato di utilizzare in passato per scopi universitari o personali. Conoscendo inoltre anche C++ e Java, l'utilizzo di C# non mi ha creato particolari problemi, se non il dover imparare le poche differenze sintattiche.
Per quanto riguarda invece gli altri strumenti, ho avuto modo di apprenderli durante lo stage. Nessuno si è rilevato particolarmente difficile da utilizzare, anche se alcuni hanno richiesto più tempo per essere compresi a fondo. Ad esempio il routing di Angular non mi è stato di immediata comprensione, ma dopo averlo utilizzato per un po' di tempo ho iniziato a capirne il funzionamento. Anche l'apprendimento completo dei molti concetti del Domain-Driven Design ha richiesto parecchie ore di studio. La parte più difficile è stata imparare a utilizzare la libreria RxJS in modo corretto capendo completamente il funzionamento dei metodi utilizzati.
Anche l'utilizzo della libreria aziendale all'inizio è stato un po' ostico, in quanto senza la completa comprensione del DDD e dei pattern utilizzati non mi era chiarissimo, ma con il tempo e controllando il codice sorgente non ha richiesto particolari sforzi.
Come detto prima, alcuni si sono rivelati semplici e molto utili: Fork ad esempio è stato una piacevole scoperta, in quanto la sua [G]GUI è molto semplice e intuitiva, ed è un ottima alternativa ad altri client Git, soprattutto a quelli integrati negli IDE.

5.2 Miglioramenti

Durante lo sviluppo il cliente indicava già eventuali miglioramenti e/o nuove funzionalità da aggiungere.
In ogni caso il primo miglioramento che salta all'occhio è sicuramente la parte grafica. Nonostante l'utilizzo di librerie come Bootstrap, Kendo e FontAwesome che forniscono componenti grafici già pronti, il risultato finale non è dei migliori. Questo è dovuto principalmente al mio poco gusto nel design in generale e al mio pensiero che la parte grafica sia secondaria rispetto alla parte funzionale. Mi è stato fatto notare però durante il tirocinio che un buon design è fondamentale per un buon prodotto, visto che è la prima cosa che l'utente vede.
Un altro miglioramento da fare è la gestione degli utenti: attualmente non c'è nessun tipo di restrizione per gli utenti che, una volta effettuato l'accesso, possono effettuare qualsiasi operazione. Per ora non è un problema, visto che il sistema non è ancora in uso nonostante una versione sia già consegnata al cliente.
L'ultimo miglioramento immediato è la visualizzazione dello storico degli oggetti rappresentati nel sistema. Attualmente infatti, grazie al sistema degli eventi descritto nel capitolo 4, vengono automaticamente salvate tutte le operazioni applicate su un aggregate root, quindi le informazioni per poterlo fare a livello di database sono già presenti. Non c'è però nessuna interfaccia che permetta agli utenti di visualizzare le vecchie versioni degli oggetti.

5.3 Valutazione personale

Questa esperienza è stata incredibilmente formativa, sia dal punto di vista tecnico che personale. Grazie alle molte tecnologie utilizzate ho potuto ampliare di molto le mie conoscenze e imparare nuovi concetti. Non è stato il mio primo tirocinio in questo settore ma è stata la prima volta che ho potuto lavorare nell'ambito informatico in modo professionale e con un progetto di una certa complessità da consegnare a un cliente vero e proprio. Anche l'utilizzo di strumenti come Azure DevOps e la review del codice sono state esperienze nuove per me, che mi hanno permesso di capire come sarebbe il lavoro in un team vero e proprio.
Ho avuto modo inoltre di imparare quali sono i miei limiti e quali sono le mie capacità, come ad esempio il poco gusto nel design.
Per concludere è stata un esperienza che mi ha permesso di crescere dal punto di vista professionale e personale, grazie anche all'ambiente di lavoro e ai colleghi che mi hanno aiutato durante il periodo di tirocinio.

Glossario

Board Una *board* è una rappresentazione visiva del lavoro che che il team di sviluppo deve svolgere, con l'obiettivo di portare chiarezza ed efficenza nel lavoro. Consiste in una lavagna in cui vengono rappresentati i compiti da effettuare e in quale stato sono attualmente. I vari compiti sono divisi per colonne che rappresentano appunto lo stato di avanzamento del compito. In genere i task vengono spostati verso destra quando c'è stato un avanzamento. 2, 61

classe generica Una *classe generica* è una classe a cui viene aggiunto un parametro di tipo, per poterla utilizzare con tipi diversi. 53, 61

Container Con il termine *container* in informatica si intende un pacchetto software leggero e autonomo che include tutto il necessario per eseguire un applicazione. Sono utili per eseguire e portabilizzare un programma. 1, 3, 61

DevOps Con il termine *DevOps* si intende una metodologia di sviluppo software che combina sviluppo (*development*, contratto in dev) e operazioni (*operations*, contratto in Ops). Prevede comunicazione frequente, collaborazione e integrazione. Tutto questo impone modifiche più frequenti al codice e un utilizzo più dinamico dell'infrastruttura fra i team operativi e di sviluppo. 2, 61

Diagramma E-R Un *diagramma E-R* è uno schema concettuale che permette di rappresentare le entità e le relazioni di un database relazionale. 35, 61

enumeratori Un' *enumeratore* è uno strumento che permette di definire un tipo di dato che può assumere solo valori specifici.. 41, 61

Git *Git* è un software gratis e open source per il controllo di versione distribuito. Il controllo di versione (*versioning*) è un metodo con cui tener traccia delle modifiche apportate a un progetto. Fornisce agli utenti gli strumenti per monitorare i cambiamenti effettuati in ogni versione, per risalire a interventi che potrebbero aver causato malfunzionamenti o errori. Permette anche di avere più versioni dello stesso progetto, ad esempio per averne una per la produzione e una per il rilascio. 2, 59, 61

GUI La *GUI* (Graphical User Interface), o interfaccia grafica, è il sistema che interagisce direttamente con l'utente. Permette di evitare l'utilizzo dell'interfaccia a linea di comando (CLI), che è più complessa e meno intuitiva. 59, 61

HTTP *HTTP* (HyperText Transfer Protocol) è il protocollo più comune per la comunicazione client-server. È basato sul concetto di risorse e di URI (Uniform Resource Identifier). Una risorsa è un oggetto identificabile univocamente e compito dell'URI è identificarla. . 1, 61

Integrated Development Environment Un *Integrated Development Environment* (Ambiente di Sviluppo Integrato) è un software che offre ai suoi utenti un ambiente per lo sviluppo, il testing, il debug e l'escuzione di applicazioni. In genere viene fornito come un pacchetto che presenta tutti gli strumenti necessari allo sviluppatore, come ad esempio un editor di codice sorgente, un compilatore, uno strumento per l'esecuzione automatica e un *debugger*. Spesso includono funzionalità avanzate come il supporto per il refactoring del codice, la gestione del versionamento e l'integrazione con altri strumenti di sviluppo. 2, 61

Log In informatica un *log* è un file dove vengono memorizzate automaticamente le operazioni e gli errori che si sono verificati su un sistema. È utile ad esempio per il monitoraggio del lavoro svolto dal programma, in modo da individuare eventuali problemi da dover risolvere o per vedere l'efficenza rispetto a determinate richieste. 62

Modale Un *modale* è una finestra figlia della principale che viene di solito utilizzata per comunicare qualcosa all'utente, visto che ne attira quasi sicuramente l'attenzione. Spesso per tornare alla principale è richiesto di interagirvi. 33, 62

navigation property Una *navigation property* è una proprietà di una entità che permette di accedere ad altre entità correlate, come ad esempio con una chiave esterna. 55, 62

Object Oriented Programming La *programmazione orientata agli oggetti* (in inglese *Object Oriented Programming* e abbreviata in *OOP*) è un paradigma di programmazione che sfrutta appunto oggetti, cioè entità definite e riutilizzabili che contengono attributi e metodi.. 50, 62

override L'*override* consiste nel ridefinire in una classe figlia un metodo ereditato da una classe padre, in modo da avere un comportamento differente in base a quale oggetto viene utilizzato. 50, 62

Query In informatica con il termine *query* si indica un comando scritto per effettuare un operazione su una base di dati. Ci sono diversi linguaggi che le query possono utilizzare, dipendenti soprattuto dal tipo di database che viene utilizzato. Il più utilizzato è SQL ossia Structured Query Language. 34, 62

repository In informatica con il termine *repository* è un archivio in cui vengono raccolti e conservati dati e informazioni corredati da descrizioni (*metadati*) in formato digitale. Rappresentano l'equivalente elettronico di una biblioteca. 2, 62

responsive Il termine *responsive* se attribuito a un'interfaccia indica che essa è in grado di adattarsi automaticamente al dispositivo sul quale si trova, evitando la necessità dell'utente di dover ridimensionare i contenuti. 32, 62

REST *REST* è uno stile architetturale per l'interazione tra componenti di un sistema distribuito. Esso definisce un insieme di vincoli per definire un applicazione RESTful ossia: client-server, stateless, cacheable, layered system, code on demand, uniform interface. 1, 31, 35, 62

Software Con *software* si intende l'insieme delle procedure e delle istruzioni eseguibili in un sistema di elaborazione dati; si contrappone ad *hardware* che corrisponde alla parte tangibile che si occupa di eseguirlo fisicamente. 2–4, 40, 62

URL L'*URL* (Unique Resource Locator) è una stringa che identifica univocamente una risorsa in una rete. In genere è utilizzato per indicare risorse web. 41, 62

Use case Gli *use case* individuano appunto dei casi d'uso di un sistema per definirli meglio durante l'analisi delle funzionalità di un software, Devono essere il più elementari possibile e sono ciò che il sistema terminato dovrà permettere di fare. 6, 8, 13, 18, 21, 24, 62

Bibliografia

Siti web consultati

Angular Routing. URL: https://angular.io/guide/routing-overview.

AutoMapper. URL: https://automapper.org/.

Bootstrap. URL: https://getbootstrap.com/.

CQRS. URL: https://learn.microsoft.com/it-it/azure/architecture/patterns/cqrs.

EFCore. URL: https://learn.microsoft.com/en-us/ef/core/.

Jaspersoft Studio. URL: https://community.jaspersoft.com/files/file/19-jaspersoft%C2%AE-studio-community-edition/.

LINQ. URL: https://learn.microsoft.com/it-it/dotnet/csharp/linq/.

Microsoft Azure DevOps. URL: https://azure.microsoft.com/it-it/products/devops.

ngx-colorpicker. URL: https://www.npmjs.com/package/ngx-color-picker.

ngx-translate. URL: https://www.npmjs.com/package/@ngx-translate/core.

Portainer. URL: https://www.portainer.io/.

RestSharp. URL: https://restsharp.dev/.

RxJS. URL: https://rxjs.dev/.