



byteyourdreams.swe@gmail.com

Specifica Tecnica

Informazioni documento

Redattore	A. Mio L. Zanesco A.M. Margarit Y. Huang O.F. Stiglet L. Albertin
Verificatore	A.M. Margarit L. Albertin Y. Huang O.F. Stiglet A. Mio L. Zanesco
Destinatari	Byte Your Dreams T. Vardanega R. Cardin

Registro delle modifiche

Versione	Data	Autore	Verificatore	Dettaglio
1.0.0	25/04/2025		L. Albertin	Approvazione finale
0.6.0	14/04/2025	L. Albertin	L. Zanesco	redazione sez. Tracciamento dei requisiti
0.5.0	14/04/2025	L. Zanesco	L. Albertin	redazione sez. Architettura: Struttura del sistema
0.4.0	10/04/2025	L. Albertin	L. Zanesco	redazione sez. Architettura: architettura di dettaglio
0.3.0	14/03/2025	L. Zanesco	L. Albertin	redazione sez. Architettura: architettura logica, architettura di deployment e componenti architetturali
0.2.0	13/03/2025	L. Albertin	L. Zanesco	redazione sez. Architettura: logica del prodotto
0.1.0	11/03/2025	L. Albertin	L. Zanesco	sez. Introduzione e Tecnologie



Indice

Byte Your Dreams

Contents

1	Introduzione	6
1.1	Scopo del documento	6
1.2	Scopo del progetto	6
1.3	Glossario	6
1.4	Riferimenti	6
1.4.1	Riferimenti normativi	6
1.4.2	Riferimenti informativi	6
2	Tecnologie	7
2.1	Docker	7
2.1.1	Docker images	7
2.2	Linguaggi e formato dati	7
2.2.1	Python	7
2.2.1.1	Versione:	8
2.2.1.2	Documentazione:	8
2.2.1.3	Utilizzo nel progetto	8
2.2.1.4	Librerie o framework	8
2.2.2	TypeScript	9
2.2.2.1	Versione:	9
2.2.2.2	Documentazione:	9
2.2.2.3	Utilizzo nel progetto	9
2.2.2.4	Librerie o framework	9
2.2.3	HTML	9
2.2.3.1	Utilizzo nel progetto	9
2.2.4	CSS	9
2.2.4.1	Utilizzo nel progetto	10
2.2.5	SQL	10
2.2.5.1	Utilizzo nel progetto	10
2.3	SQL (Structured Query Language)	10
2.3.0.1	Utilizzo nel progetto	10
2.3.1	YAML (YAML Ain't Markup Language)	10



2.3.1.1	Utilizzo nel progetto	10
2.3.2	JSON (JavaScript Object Notation)	10
2.3.2.1	Utilizzo nel progetto	10
2.4	Database e servizi	10
2.4.1	Supabase	10
2.4.1.1	Versione	10
2.4.1.2	Documentazione	10
2.4.1.3	Servizi e vantaggi di Supabase	11
2.4.1.4	Approfondimento Database PostgreSQL di Supabase	11
2.4.1.5	Utilizzo nel progetto	11
3	Architettura	12
3.1	Logica del prodotto	12
3.2	Architettura logica	13
3.3	Architettura di deployment	14
3.4	Componenti architetturali	14
3.4.1	Assemblaggio componenti e deployment	14
3.5	Architettura di dettaglio	15
3.5.1	Architettura della generazione di una risposta	15
3.5.2	Architettura dell'aggiornamento del database	16
3.5.3	Architettura fine aggiornamento del database	18
3.5.4	Architettura dell'inserimento associazione prodotto - pdf nel database	19
3.5.5	Architettura dell'inserimento delle FAQ nel database	20
3.5.6	Architettura dell'inserimento dei file nel database	21
3.5.7	Architettura dell'inserimento dei prodotti nel database	23
3.5.8	Architettura dell'upload dei file nel database	24
3.6	Struttura del sistema	25
3.6.1	Frontend	25
3.6.1.1	Componenti principali	26
3.6.1.2	Servizi	27
3.6.1.3	Area amministratore	30
3.6.1.4	Design pattern frontend	32
3.6.2	Backend	32
3.6.2.1	Componenti principali dello Scraper	33
3.6.2.2	Componenti principali delle Supabase functions	38
3.6.2.3	Design pattern backend	47
3.6.3	Database	48
3.6.3.1	Entità principali	48
3.6.3.2	Autenticazione e chat	49



4 Tracciamento dei requisiti	49
4.1 Tabella dei requisiti funzionali	49
4.2 Riepilogo requisiti funzionali	52



1 Introduzione

1.1 Scopo del documento

Questo documento fornisce una descrizione dettagliata dell'*architettura_G* del *sistema_G*, delle scelte progettuali effettuate e delle tecnologie utilizzate.

Lo scopo di tale documento è di fornire linee guida per l'*attività_G* di codifica e di manutenzione, oltre che di motivare le scelte progettuali effettuate.

1.2 Scopo del progetto

L'obiettivo di **Vimar GENIALE** è di sviluppare un *applicativo_G* che permette agli *installatori_G* di richiedere facilmente, tramite linguaggio naturale, informazioni tecniche dei prodotti Vimar presenti sul sito ufficiale aziendale (*vimar.com*).

L'*applicativo_G* sarà fruibile attraverso un'*applicativo_G* web con interfaccia *responsive_G*, dove gli *installatori_G* potranno porre domande in linguaggio naturale per ottenere risposte riguardanti le caratteristiche e altri dettagli tecnici dei prodotti. All'interno dell'*applicativo_G* è presente un'area riservata agli *admin_G* dove sarà possibile visualizzare una *dashboard_G* con le statistiche dell'andamento dell'*applicativo_G*, e dove sarà possibile avviare il processo di aggiornamento dati. Tale processo preleva le informazioni riguardanti i prodotti dal sito web aziendale e le salva all'interno di un *database_G*.

Le risposte alle domande degli *installatori_G* saranno generate tramite un *LLM_G* interno e attraverso un processo di *RAG_G*.

1.3 Glossario

La documentazione contiene riferimenti al documento **Glossario v2.0.0**, all'interno del quale vengono esposti i significati di tutti quei termini troppo specifici o potenzialmente ambigui presenti all'interno del documento.

I termini che fanno riferimento al *Glossario* sono contrassegnati da una G posta a pedice del termine stesso.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- **Capitolato C2 - Vimar GENIALE:**
 - <https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C2.pdf> (Consultato: 11/03/2025)
- **Regolamento Progetto:**
 - <https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/PD1.pdf> (Consultato: 11/03/2025)
- **Norme di progetto v2.0.0.**

1.4.2 Riferimenti informativi

- **Analisi dei requisiti v2.0.0**
- **Glossario v2.0.0**
- **Documentazione Angular:**
 - <https://angular.dev/overview> (Consultato: 11/03/2025)



- **Documentazione Chart.js:**
 - <https://www.chartjs.org/docs/latest/> (Consultato: 11/03/2025)
- **Documentazione Scrapy:**
 - <https://docs.scrapy.org/en/2.12/> (Consultato: 11/03/2025)
- **Documentazione Scrapyd**
 - <https://scrapyd.readthedocs.io/en/stable/> (Consultato: 11/03/2025)
- **Documentazione Supabase:**
 - <https://supabase.io/docs> (Consultato: 11/03/2025)
- **Documentazione Docker:**
 - <https://docs.docker.com/> (Consultato: 11/03/2025)
- **Documentazione Docker Compose:**
 - <https://docs.docker.com/compose/> (Consultato: 11/03/2025)
- **Documentazione Python:**
 - <https://docs.python.org/release/3.12.0/> (Consultato: 11/03/2025)
- **Documentazione Pytest:**
 - <https://docs.pytest.org/en/7.1.x/> (Consultato: 11/03/2025)
- **Documentazione TypeScript:**
 - <https://www.typescriptlang.org/docs> (Consultato: 11/03/2025)

2 Tecnologie

All'interno di questa sezione sono definite le tecnologie e gli strumenti impiegati per lo sviluppo del progetto **Vimar GENIALE**.

Verranno descritte le tecnologie, i linguaggi di programmazione, i *framework_G* e le librerie utilizzate per lo sviluppo del *progetto_G*.

2.1 Docker

Per garantire ambienti consistenti e riproducibili durante lo sviluppo, il *testing_G* e il rilascio del prodotto sono stati utilizzati *container Docker_G*.

2.1.1 Docker images

Di seguito sono riportate le immagini *Docker_G* utilizzate per lo sviluppo del *progetto_G*:

2.2 Linguaggi e formato dati

2.2.1 Python

Linguaggio di programmazione ad alto livello, orientato agli oggetti, interpretato e multiparadigma.



Servizio	Immagine Docker	Descrizione
kong	kong:2.8.1	API Gateway
auth	supabase/gottrue:v2.158.1	Servizio di autenticazione
rest	postgrest/postgrest:v12.2.0	REST API per PostgreSQL
realtime	supabase/realtime:v2.30.34	Aggiornamenti in tempo reale
minio	minio/minio	Servizio per lo storage
minio-createbucket	minio/mc	Creazione bucket Minio
storage	supabase/storage-api:v1.11.13	Servizio di storage
imgproxy	darthsim/imgproxy:v3.8.0	Trasformazione immagini
meta	supabase/postgres-meta:v0.83.2	API gestione PostgreSQL
functions	supabase/edge-runtime:v1.58.3	Runtime per edge functions
db	supabase/postgres:15.1.1.78	Database PostgreSQL
vector	timberio/vector:0.28.1-alpine	Logging e aggregazione dati
supervisor	supabase/supervisor:1.1.56	Database connection pooler
scrapy	(costruito da ../Scraper)	Servizio di scraping
ollama	ollama/ollama:0.5.13	Runtime AI (GPU supportata)
web app	angular-app	Applicativo web

Table 1: Elenco delle immagini Docker utilizzate nel progetto Vimar GENIALE

2.2.1.1 Versione:

Versione utilizzata: 3.12

2.2.1.2 Documentazione:

<https://docs.python.org/release/3.12.0/> (Consultato: 11/03/2025)

2.2.1.3 Utilizzo nel progetto

- Creazione dello *scraper*_G per l'estrazione dei dati dai siti web *vimar.com*.

2.2.1.4 Librerie o framework

- **Scrapy:**
 - **Documentazione:** <https://docs.scrapy.org/en/2.12/> (Consultato: 11/03/2025)
 - **Versione:** 2.12
 - *Framework*_G open-source_G in Python per il web scraping.
- **Pytest**
 - **Documentazione:** <https://docs.pytest.org/en/7.1.x/> (Consultato: 11/03/2025)
 - **Versione:** 7.1
 - *Framework*_G di testing per Python che consente la scrittura di *test*_G chiari e concisi, attraverso una semplice sintassi.
- **supabase-py**
 - **Documentazione:** <https://supabase.io/docs/reference/python/supabase-client> (Consultato: 11/03/2025)
 - **Versione:** 2.13
 - *Client*_G Python per l'interazione con il servizio *Supabase*_G.



2.2.2 TypeScript

Linguaggio di programmazione *open-source*, sviluppato da Microsoft, che estende JavaScript aggiungendo il supporto ai tipi statici, migliorando la gestione degli errori, e aggiungendo maggior modularità.

2.2.2.1 Versione:

Versione utilizzata: 5.3

2.2.2.2 Documentazione:

<https://www.typescriptlang.org/docs/> (Consultato: 11/03/2025)

2.2.2.3 Utilizzo nel progetto

- Sviluppo del frontend dell'applicativo web in *Angular*.
- Sviluppo Edge-functions fornite da *Supabase*.

2.2.2.4 Librerie o framework

• Angular

- **Documentazione:** <https://angular.dev/overview> (Consultato: 11/03/2025)
- **Versione:** 19.0
- *Framework* per lo sviluppo di applicazioni web basate su componenti.

• supabase-js

- **Documentazione:** <https://supabase.io/docs/reference/javascript/supabase-client> (Consultato: 11/03/2025)
- **Versione:** 2.38
- *Client* JavaScript/TypeScript per l'interazione con il servizio *Supabase*.

• Chart.js

- **Documentazione:** <https://www.chartjs.org/docs/latest> (Consultato: 11/03/2025)
- **Versione:** 4.4.8
- Libreria compatibile con i più utilizzati *framework* Javascript (come Angular), utilizzata per la creazione di diverse tipologie di grafici e di plugin.

• Deno

- **Documentazione:** <https://docs.deno.com/> (Consultato: 11/03/2025)
- **Versione:** 1.45.2
- Runtime *open source* per TypeScript e JavaScript.

2.2.3 HTML

Linguaggio di *markup* per la creazione di pagine web.

2.2.3.1 Utilizzo nel progetto

- Creazione della struttura dei componenti web.

2.2.4 CSS

Linguaggio di stile per la definizione della presentazione di pagine web.



2.2.4.1 Utilizzo nel progetto

- Creazione stile dei componenti dell'*applicativo_G* web.

2.2.5 SQL

Linguaggio di programmazione *standard_G* per la gestione e la manipolazione di *basi di dati_G*.

2.2.5.1 Utilizzo nel progetto

- Creazione, gestione e interragoazione del *database_G Geniale_G*.

2.3 SQL (Structured Query Language)

Linguaggio di programmazione per l'archiviazione e l'elaborazione di informazioni in un *database_G* relazionale.

2.3.0.1 Utilizzo nel progetto

- Configurazione *Docker_G*.
- Configurazione *kong_G*.
- Configurazione *Github_G workflow_G* per *Continuous Integration_G*.

2.3.1 YAML (YAML Ain't Markup Language)

Formato di serializzazione di dati leggibile dall'uomo, utilizzato per rappresentare dati strutturati.

2.3.1.1 Utilizzo nel progetto

- Gestione e interrogazione del *database_G PostgreSQL_G*.

2.3.2 JSON (JavaScript Object Notation)

Formato basato su testo per l'archiviazione e lo scambio di dati in modo leggibile e modificabile dall'uomo e dal computer.

Tale formato trova impiego in diversi contesti, tra cui lo sviluppo web, le *API_G* di servizi web e lo scambio di dati tra applicazioni.

2.3.2.1 Utilizzo nel progetto

- Risposte all'interno della funzione *Deno.serve()* all'interno di ciascun file *index.ts* relativo alle *functions_G* di *Supabase_G*.

2.4 Database e servizi

2.4.1 Supabase

Alternativa *open-source_G* a *Firebase* che fornisce un'infrastruttura *backend_G* per lo sviluppo di *applicazioni_G* moderne. *Supabase_G* adotta il modello di servizio *Backend-as-a-service_G* (BaaS), il quale offre soluzioni già pronte semplificando notevolmente il lavoro degli sviluppatori sul lato del *backend_G*.

2.4.1.1 Versione

Versione utilizzata:

2.4.1.2 Documentazione

<https://supabase.com/docs> (Consultato: 31/03/2025)



2.4.1.3 Servizi e vantaggi di Supabase

I principali servizi e funzionalità offerti da *Supabase_G* sono:

- **Database PostgreSQL:** *database_G PostgreSQL_G* completamente gestito, che fornisce agli sviluppatori un'infrastruttura scalabile e affidabile per l'archiviazione e la gestione dei dati delle loro *applicazioni_G*. Include funzionalità avanzate come controllo degli accessi basato su *Row-Level Security* (RLS) e supporto per estensioni *PostgreSQL_G*, garantendo efficienza e flessibilità nello sviluppo;
- **Autenticazione:** servizio che permette l'autenticazione degli *utenti_G*, tramite diverse opzioni, come l'accesso tramite e-mail e password, l'accesso tramite *provider_G* esterni, e persino l'accesso tramite *magic link*. Questo permette agli sviluppatori di implementare facilmente e in modo sicuro i *processi_G* di gestione degli *utenti_G*;
- **Archiviazione:** Servizio di storage che permette di archiviare e gestire file multimediali e *dati_G* di grandi dimensioni come immagini, video e altri file all'interno dell'*applicazione_G*;
- **Edge Functions:** le *Edge Functions_G* di *Supabase_G*, equivalenti alle *funzioni serveless_G*, vengono attivate da eventi ed eseguite nel *cloud_G*. Questo permette agli sviluppatori di scrivere e distribuire facilmente codice personalizzato senza doversi preoccupare dell'infrastruttura sottostante;
- **Real Time:** Servizio che permette la sincronizzazione dei *dati_G* in tempo reale. Questo è fondamentale per *applicazioni_G* che necessitano di aggiornamenti immediati e interattività in tempo reale;
- **AI & Vectors:** *Toolkit_G open-source_G* per lo sviluppo di *applicazioni_G* basate sull'*AI_G* utilizzando *Postgres_G* e *pgvector_G*. Questo *toolkit_G* facilita la memorizzazione dei vettori all'interno del *database_G* e l'interazione con servizi come *LangChain_G*, *OpenAI_G* e *Hugging Face_G*. Permette inoltre diverse tipologie di ricerca basata sulla similarità vettoriale come la Semantic Search, Keyword Search e la Hybrid Search.

2.4.1.4 Approfondimento Database PostgreSQL di Supabase

Ogni *progetto_G* *Supabase_G* è dotato di un *database_G PostgreSQL_G*, un potente sistema di *database relazionale_G open-source_G*, conosciuto per la sua affidabilità, robustezza delle funzionalità e performance elevate.

La particolarità di questo *database_G* è la sua totale portabilità, ovvero rende possibile e facilita la migrazione di *dati_G* da e verso altri *database_G PostgreSQL_G*. Questa essenziale caratteristica assicura che i *dati_G* non vengano mai bloccati all'interno di un *sistema_G* proprietario.

2.4.1.5 Utilizzo nel progetto

All'interno di **Vimar GENIALE**, *Supabase_G* svolge un ruolo centrale, costituendo l'intero *backend_G*.

In particolare, viene implementato:

- Il *sign in anonimo_G*, messo a disposizione dal servizio di autenticazione di *Supabase_G*. Questa tipologia di autenticazione permette agli *utenti_G* di accedere al sistema senza l'immissione di credenziali personali.
- Il *log in_G* per consentire agli amministratori di accedere, inserendo e-mail e password, alla *dashboard_G*;
- Il servizio di storage per immagazzinare i manuali scaricati dal sito web dell'azienda *proponente_G*;
- Le *Edge Functions_G* per implementare le funzioni richieste dal prodotto. Sono state create *Edge Functions_G* per il *chunking_G* dei documenti, per la generazione della risposta attraverso l'*LLM_G* e per l'elaborazione dei *dati_G* necessaria per una loro successiva visualizzazione all'interno della *dashboard_G*;



- Il servizio di real time per garantire un aggiornamento in tempo reale dei $dati_G$ presenti all'interno del $database_G$.
- Il modulo di $AI & Vectors$ viene utilizzato per la creazione di una colonna all'interno del $database_G$ adibita a immagazzinare i diversi vettori risultanti dall' $embedding_G$ dei diversi chunk dei documenti. Inoltre viene sfruttato per effettuare la ricerca di similarità tra l' $embedding_G$ della domanda posta dall' $utente_G$ e l' $embedding_G$ di tutti i chunk contenuti nel $database_G$.

3 Architettura

Durante l' $attività_G$ di progettazione si è deciso di non adottare un pattern architettonico univoco per tutta l' $architettura_G$ sottostante ma il $sistema_G$ è stato convenzionalmente diviso a livello logico e strutturale tra:

- $frontend_G$, ovvero la parte client del $sistema_G$, sviluppata attraverso il $framework_G Angular_G$ e $TypeScript$;
- $backend_G$, sfruttando i diversi servizi e funzionalità messi a disposizione da $Supabase_G$.

La comunicazione tra le due parti avviene tramite $Kong_G$, che rendirizzare le chiamate API_G effettuate dall' $applicativo_G$ web al corretto servizio di $Supabase_G$. Nelle seguenti sezioni verranno approfondite le diverse componenti, illustrando i pattern architettonici e le scelte di design adottate dal $team_G$ durante l' $attività_G$ di progettazione.

3.1 Logica del prodotto

L' $applicazione_G$ web **Vimar GENIALE** permette di richiedere informazioni riguardo i prodotti Vimar, sfruttando un' LLM_G (*Large Language Model*) che interroga un $database_{relazionale}_G$ contenente dati tecnici e documenti dei prodotti di **Vimar S.p.A.**

Le informazioni presenti nel $database_G$ vengono estratte attraverso un processo di $web scraping_G$ del sito web aziendale, avviato su richiesta dell'amministratore di sistema. Durante questo processo, ogni volta che un nuovo prodotto viene estratto e salvato, i relativi documenti vengono scaricati, memorizzati nel $database_G$ e suddivisi in unità informative più piccole, dette $chunk_G$, successivamente trasformate in vettori numerici.

Quando un $utente_G$ accede alla pagina web di **Vimar GENIALE**, viene eseguita un'autenticazione anonima che gli consente di richiedere informazioni. Nel momento in cui l' $utente_G$ invia una domanda tramite l'interfaccia web, questa viene inserita nel $database_G$, attivando tramite $trigger_G$ un'apposita $edge function_G$, che esegue in sequenza le seguenti operazioni:

- Analizza la domanda dell'utente e la classifica in base al tipo di richiesta (generale, tecnica o differenza tra prodotti);
- Recupera le informazioni del prodotto;
- Recupera i $chunk$ più pertinenti tramite una ricerca di similarità ibrida;
- Costruisce, tramite i dati recuperati, un prompt da inviare all' LLM_G ;
- Interroga l' LLM_G inviando il prompt generato;
- Inserisce la risposta ottenuta all'interno del $database_G$.

Una volta che la risposta viene memorizzata, essa viene notificata e visualizzata all'utente in tempo reale tramite l'utilizzo di $web sockets_G$. Nel caso in cui l' $utente_G$ non fosse soddisfatto della risposta ricevuta, potrà inviare un $feedback_G$, che verrà memorizzato all'interno del $database_G$ e utilizzato per migliorare le prestazioni del $sistema_G$.



Qualora la domanda dell'utente fosse fuori contesto, facesse riferimento a un prodotto non presente nel *database_G*, o si verificassero errori durante l'estrazione delle informazioni e/o la generazione della risposta, l'*utente_G* riceverà un messaggio generico che lo avviserà dell'errore, invitandolo a riprovare successivamente o a riformulare la domanda.

Dopo l'autenticazione anonima, l'*utente_G* visualizzerà lo storico delle conversazioni salvate. Selezionando una di quest'ultime conversazione, potrà consultare tutte le domande poste e le risposte ricevute. Inoltre, potrà creare una nuova conversazione (se non ha raggiunto il numero massimo consentito) oppure eliminarne una già esistente.

L'*applicazione_G* web include una sezione riservata agli *admin_G*, accessibile previo login. In quest'area, l'amministratore ha accesso a un cruscotto informativo contenente grafici sull'andamento dell'utilizzo dell'*applicativo_G* e può visualizzare i *feedback_G* ricevuti dagli utenti. Inoltre, può avviare manualmente il processo di *web scraping_G* per aggiornare i dati presenti nel *database_G*.

3.2 Architettura logica

L'*architettura_G* logica adottata nella realizzazione dell'*applicativo_G* è incentrata sul modello di *architettura esagonale_G*. In questo, la logica di business è indipendente dagli altri componenti del *sistema_G*, permettendo di ottenere un prodotto *software_G* facilmente testabile, *scalabile_G* e *manutenibile_G*. Gli elementi chiave che ne abilitano il corretto funzionamento sono i seguenti:

- **Controller:** contiene la *application logic_G* del *sistema_G* e si occupa della gestione delle richieste in ingresso. In particolare, ha il compito di validare i *dati_G* di tipo *DTO_G* ricevuti in input e convertirli in strutture dati compatibili con il dominio applicativo. Successivamente, il *Controller* invoca lo *Use Case* appropriato per l'esecuzione della *business logic_G*. Al termine dell'elaborazione, adatta la risposta restituita in un oggetto *DTO* e la inoltra al client;
- **Use Case:** rappresenta un caso d'uso specifico del *sistema_G* e viene implementato da un *Service*, il quale si occupa dell'esecuzione della relativa logica di business;
- **Service:** contiene la logica di business del *sistema_G*, occupandosi dell'esecuzione di operazioni specifiche e strettamente legate al dominio applicativo. I *Service* delegano le interazioni con componenti esterne ai *Port*, mantenendo così un elevato grado di separazione delle responsabilità. Operano esclusivamente su tipi di *dato_G* appartenenti al dominio, assicurando l'indipendenza della logica di business rispetto agli altri strati del *sistema_G*;
- **Port:** definisce le interfacce che i *Service* utilizzano per interagire con componenti esterni al dominio applicativo. In questo modo è possibile effettuare operazioni come il salvataggio e il recupero di dati persistenti, senza introdurre dipendenze dirette all'interno della logica di business, preservandone così l'indipendenza e la coesione;
- **Adapter:** implementa una o più interfacce definite dai *Port*, facilitando la comunicazione tra la logica di business e le tecnologie esterne utilizzate per la *persistent logic_G*. L'adapter è responsabile di adattare i *dati_G* ricevuti dalla logica di business in un tipo *Entity_G* idoneo per la persistenza, e viceversa;
- **Repository:** contiene la *persistent logic_G* del *sistema_G*, gestendo la persistenza dei *dati_G* tramite l'interazione con un *database_G* o altre forme di storage per il salvataggio e il recupero delle informazioni necessarie. I tipi di *dato_G* trattati dal repository sono gli *Entity_G*, che fungono da livello intermedio tra i tipi di business e i *dati_G* persistenti del *sistema_G*.

Le principali motivazioni per la scelta di un'*architettura esagonale_G* sono:

- **Testabilità:** migliora la testabilità della *business logic_G* sostituendo gli adapter con *stub_G* o *mock_G*;



- **Manutenibilità:** favorisce la separazione delle responsabilità dell'*applicazione_G* garantendo il mantenimento di un'*architettura_G* modulare e favorendo il riuso del codice;
- **Scalabilità:** favorisce l'adattabilità dell'*applicazione_G* alle esigenze di cambiamento. L'*applicazione_G* può essere facilmente modificata o estesa, senza influire sulla logica di business centrale. Ciò consente di adattarsi rapidamente alle nuove funzionalità richieste o di integrarsi con nuovi *sistemi_G* esterni.

3.3 Architettura di deployment

Per determinare l'*architettura di deployment_G* più adeguata all'*applicativo_G*, si è tenuto conto delle esigenze di scalabilità, manutenibilità e flessibilità richieste dal contesto operativo. Considerando la necessità di gestire in modo modulare le funzionalità del *sistema_G* e di poter evolvere singolarmente i diversi componenti, è stata adottata un'*architettura a microservizi_G*. Questa scelta consente di distribuire e scalare in maniera indipendente i diversi servizi applicativi, migliorando la robustezza e facilitando l'integrazione di nuovi moduli in futuro. Rispetto a un'*architettura monolitica_G*, il modello a microservizi permette un'organizzazione del codice più chiara e specializzata. Tale approccio, sebbene introduca una maggiore complessità architettonica, è stato ritenuto adeguato in relazione alla necessità di garantire all'azienda maggiore autonomia nell'evoluzione dell'*applicativo_G*.

Il *processo_G* di deployment del *sistema_G* è stato implementato tramite l'utilizzo di *Docker Compose_G*, al fine di offrire un ambiente di esecuzione già configurato e facilmente replicabile. Questa soluzione permette di avviare simultaneamente tutti i componenti necessari al funzionamento dell'*applicativo_G*, evitando le complessità legate a configurazioni manuali e garantendo coerenza tra gli ambienti di sviluppo, test e produzione.

3.4 Componenti architetturali

L'*architettura_G* del *sistema_G* è suddiviso nei seguenti componenti principali:

- **Frontend:** fornisce un'interfaccia grafica *utente_G* per interagire con il *sistema_G*. Si occupa di inoltrare le domande al *backend_G* e visualizzare i risultati ricevuti;
- **Backend:** gestisce l'elaborazione delle richieste degli *utenti_G*. Interagisce con il *sistema_G* di persistenza dei dati e con servizi esterni, in particolare con il *database_G* vettoriale e con l'*LLM_G*;
- **Database:** responsabile della memorizzazione della documentazione per la *RAG_G* e delle chat con i relativi messaggi.

3.4.1 Assemblaggio componenti e deployment

Le diverse componenti del *sistema_G* sono orchestrate e messe in esecuzione utilizzando *Docker Compose_G*, uno strumento che consente di definire e gestire *applicazioni_G* multi-container in modo semplice ed efficiente. Attraverso l'utilizzo di file di configurazione in formato YAML, *Docker Compose_G* permette di descrivere l'intera *architettura_G* del *sistema_G*, specificando le immagini Docker da utilizzare, le dipendenze tra i container, le variabili d'ambiente, le reti e i volumi condivisi.

Questa scelta progettuale consente di facilitare notevolmente il *processo di deployment_G*, rendendo l'avvio del *sistema_G* rapido e ripetibile, sia in ambienti di sviluppo che in ambienti di produzione. Ogni componente – *frontend*, *backend* e *database* – viene eseguito all'interno di un *container_G* dedicato, garantendo così isolamento, portabilità e una gestione modulare delle risorse. Inoltre, *Docker Compose_G* semplifica le operazioni di aggiornamento, monitoraggio e manutenzione dell'infrastruttura, contribuendo a migliorare la scalabilità e la manutenibilità complessiva del *sistema_G*.



3.5 Architettura di dettaglio

3.5.1 Architettura della generazione di una risposta

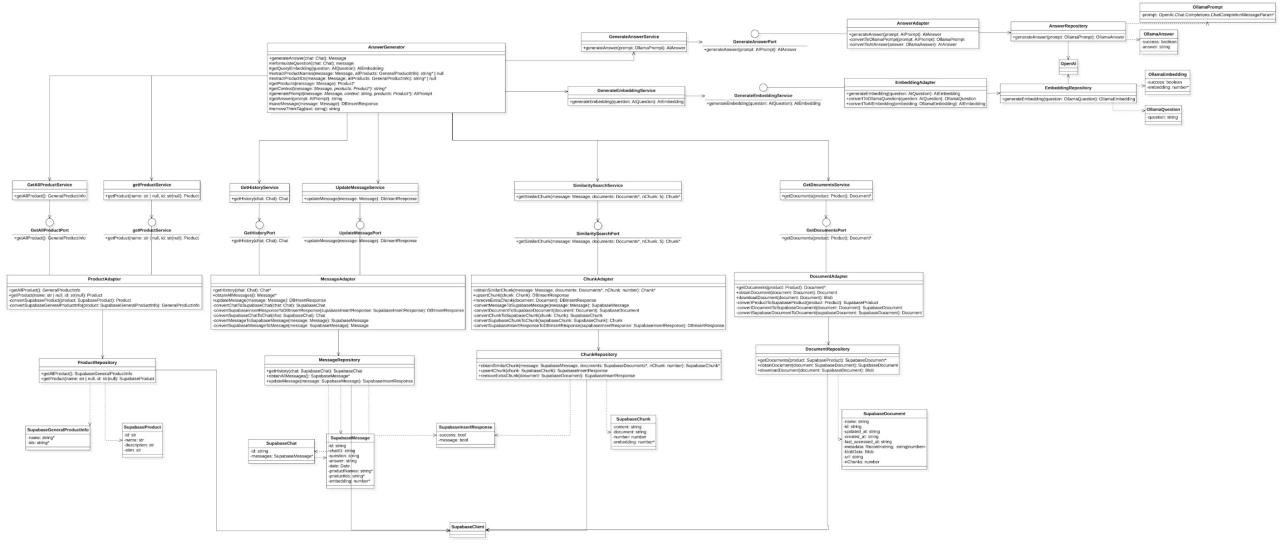


Figure 1: Architettura della generazione di una risposta

Il diagramma UML_G fornito descrive l'architettura G $backend_G$ responsabile della generazione automatica di risposte a partire dalle domande degli utenti G . L'interazione ha inizio nel $frontend_G$, che effettua una richiesta al $backend_G$, inviando i dati G utente G (messaggio e chat) in formato strutturato.

La richiesta, una volta ricevuta, viene inoltrata alla classe *AnswerGenerator*, responsabile di orchestrare i vari servizi necessari. Il primo servizio con cui *AnswerGenerator* si confronta è il *GetAllProductsService*, il cui compito è quello di recuperare l'elenco completo dei prodotti presenti nel sistema G . Questo servizio comunica tramite la porta *GetAllProductPort* con un $Adapter_G$ che, a sua volta, accede ai dati G attraverso il *ProductRepository*. Parallelamente, allo scopo di ottenere informazioni specifiche su determinati prodotti, interviene il *GetProductsService*, che tramite la propria porta *GetProductPort* consente di recuperare dettagli più mirati relativi a singoli articoli.

La gestione del contesto della conversazione è affidata al *GetHistoryService*, che ha il compito di recuperare la cronologia dei messaggi associati alla chat corrente. Questo servizio utilizza il *GetHistoryPort* e accede al database G tramite il *MessageRepository*, consentendo al sistema G di costruire risposte che tengano conto dello storico dell'interazione.

Per quanto riguarda il recupero dei contenuti documentali utili a supportare la generazione della risposta, *AnswerGenerator* si appoggia al *GetDocumentsService*. Quest'ultimo, utilizzando la porta *GetDocumentsPort*, si connette al *DocumentRepository* per ottenere i documenti pertinenti all'ambito della conversazione o dei prodotti selezionati.

Una componente essenziale del flusso è rappresentata dal *SimilaritySearchService*, incaricato di eseguire una ricerca di similarità ibrida. Questo servizio, attraverso il *SimilaritySearchPort*, accede agli $embeddings_G$ per individuare i $chunk_G$ più affini al significato del messaggio dell'utente. La comunicazione esterna avviene tramite un $Adapter_G$ dedicato, che si interfaccia con $repository_G$ di tipo vettoriale, come il *ChunkRepository*.

Successivamente, tutti i dati G raccolti vengono utilizzati dal *GenerateAnswerService*, il quale si occupa di comporre un $prompt_G$ adeguato e richiedere al modello LLM_G la generazione della risposta. Questo servizio opera mediante il *GenerateAnswerPort* e si avvale dell'*AnswerAdapter* per inviare il $prompt_G$ costruito all'*AnswerRepository*.

Infine, una volta ottenuta la risposta, *AnswerGenerator* si avvale del *UpdateMessageService* per aggiornare

il messaggio originale con il testo generato. Questo servizio, tramite la porta *UpdateMessagePort*, consente di sincronizzare l'entità messaggio con il contenuto della risposta attraverso il *MessageRepository*, garantendo la persistenza coerente della conversazione.

3.5.2 Architettura dell'aggiornamento del database

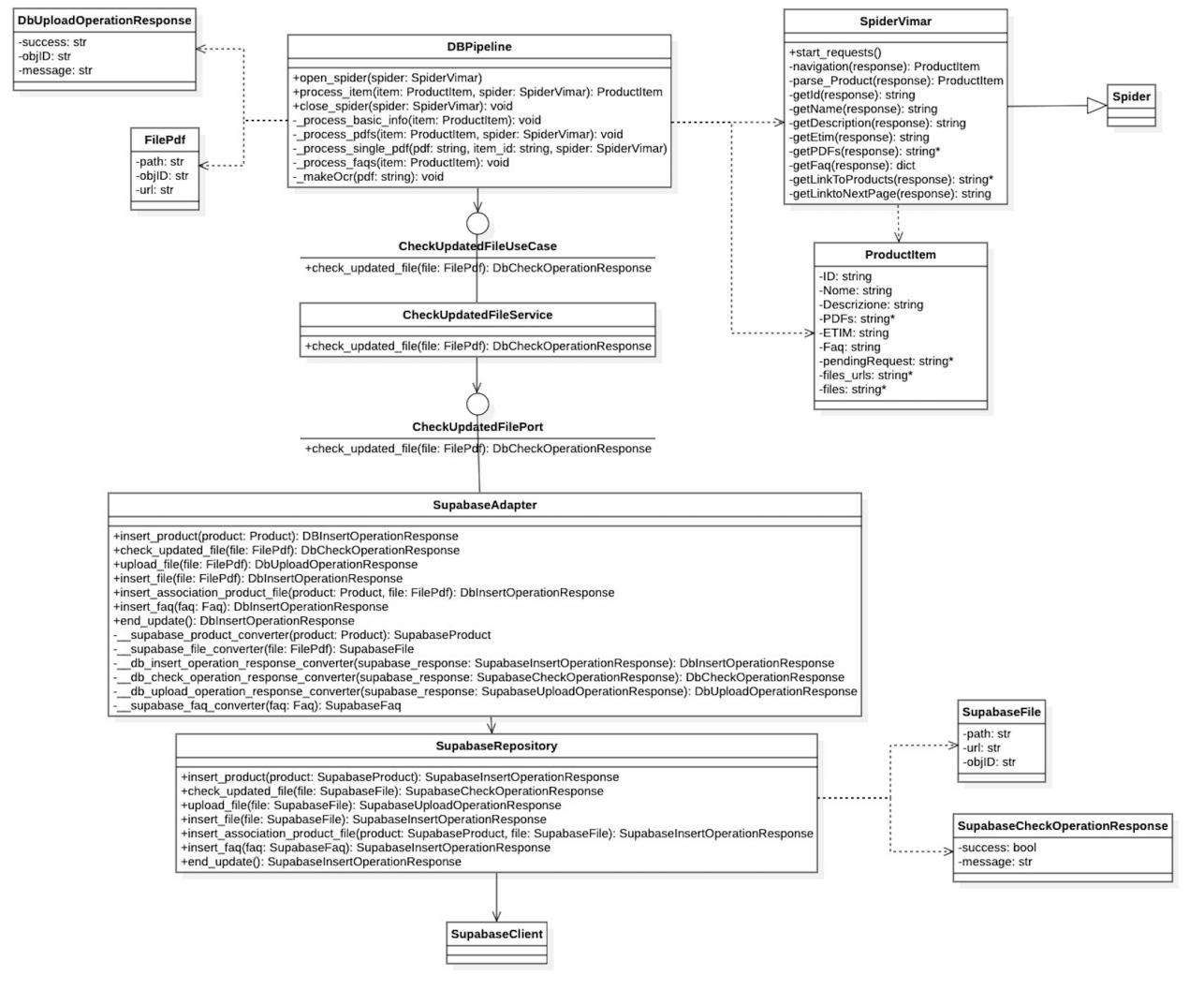


Figure 2: Architettura dell'aggiornamento del database

Il diagramma UML_G sopra presentato descrive il flusso relativo all'aggiornamento dei file all'interno del $database_G$, operazione che si basa sull'utilizzo di uno $scraper_G$ incaricato di estrarre informazioni aggiornate dal sito web aziendale. Questo $processo_G$ viene orchestrato principalmente dalla classe *DBPipeline*, la quale coordina le varie operazioni di raccolta, processione e inserimento dei $dati_G$.

Il componente *SpiderVimar* rappresenta il vero e proprio $scraper_G$. Si tratta di un modulo specializzato che, partendo dalla funzione *start_requests()*, è in grado di navigare tra le pagine del sito, estraendo progressivamente i $dati_G$ di interesse. *SpiderVimar* implementa diverse funzioni di $parsing_G$, come *parse_Product()*, che analizza la pagina di dettaglio del prodotto trasformandola in un oggetto di tipo *ProductItem*. Quest'ultimo è una semplice struttura dati che contiene attributi come ID, nome, descrizione, PDF associati, dati ETIM, FAQ_G e URL dei file.

I $dati_G$ raccolti dal $crawler_G$ vengono poi passati al *DBPipeline*, che li processa in diversi formati. Il metodo *process_item()* consente di trattare l'oggetto *ProductItem* generale, mentre metodi più specifici come *_process_pdfs()*, *_process_singl()* e *_process_faqs()* si occupano rispettivamente della gestione dei file PDF,



delle singole immagini e delle FAQ_G associate ai prodotti.

Parallelamente, il $sistema_G$ integra un meccanismo di verifica dello stato aggiornato dei file nel $database_G$. Questa logica è implementata all'interno del *CheckUpdatedFileService*, delegata dal caso d'uso *CheckUpdatedFileUseCase* e che a sua volta comunica con l'esterno tramite la porta *CheckUpdatedFilePort*. L'input fornito a questi componenti è modellato attraverso l'entità *FilePdf*, che incapsula informazioni come il path locale del file, il relativo URL e l'identificativo dell'oggetto nel $database_G$.

Tutte le operazioni di scrittura e aggiornamento sul $database_G$ vengono delegate al *SupabaseAdapter*. Questo $Adapter_G$ fornisce un'interfaccia di alto livello per operazioni come l'inserimento di prodotti, l'upload di file, la creazione di associazioni tra prodotti e file, e l'inserimento delle FAQ_G . Internamente, *SupabaseAdapter* si occupa anche della conversione tra le entità di dominio (come *Product* o *FilePdf*) e le entità persistenti specifiche per $Supabase_G$, come *SupabaseProduct* o *SupabaseFile*.

La persistenza vera e propria è gestita infine dal *SupabaseRepository*, che interagisce direttamente con il client di $Supabase_G$. *SupabaseRepository* implementa i metodi di basso livello per l'inserimento dei prodotti, il caricamento dei file, la gestione delle associazioni e l'aggiornamento delle FAQ_G , restituendo risposte standardizzate come *SupabaseInsertOperationResponse* o *SupabaseCheckOperationResponse*. Quest'ultima rappresenta l'esito delle operazioni di verifica dei file, indicando tramite un booleano di successo e un messaggio eventuali errori o conferme.



3.5.3 Architettura fine aggiornamento del database

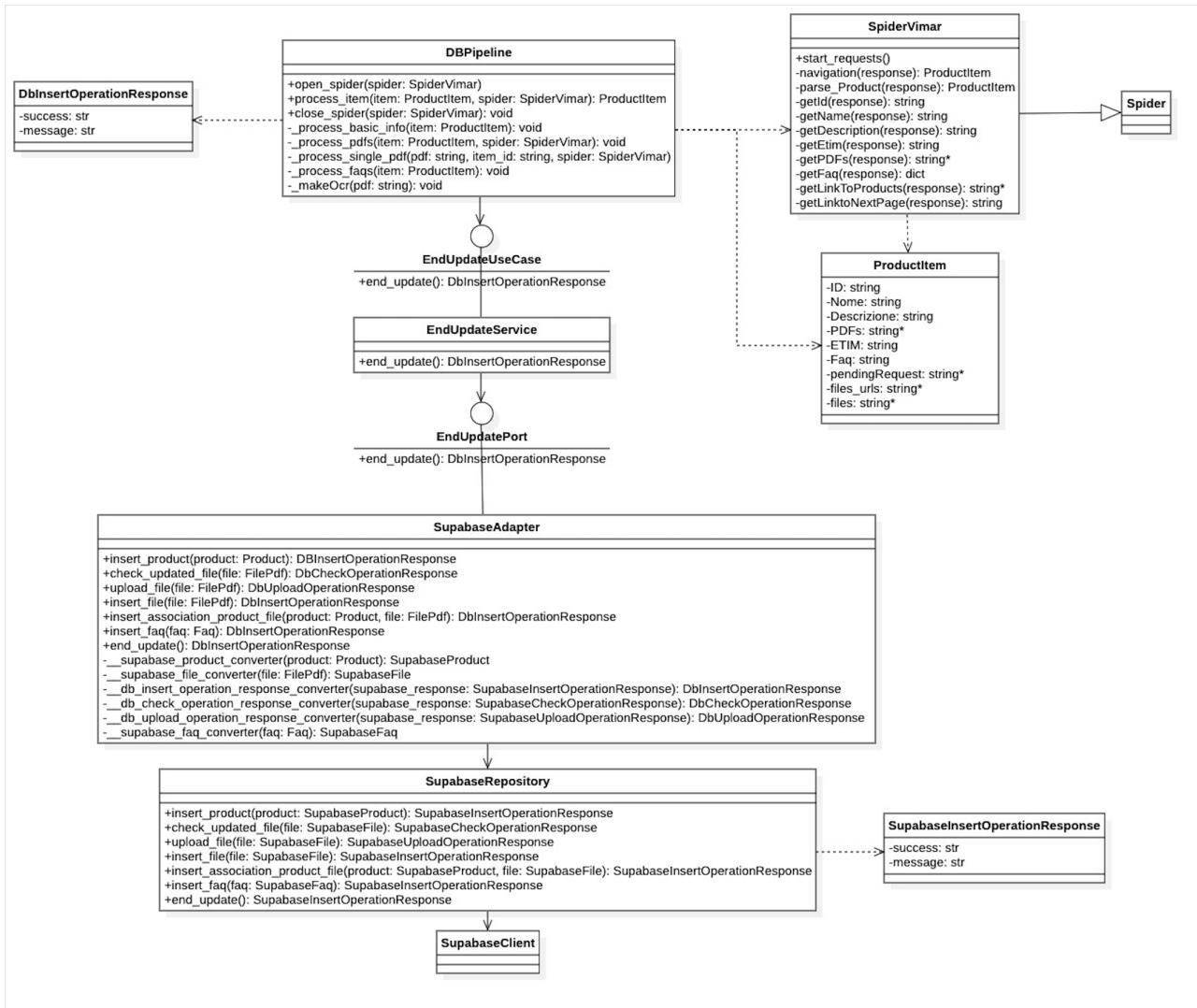


Figure 3: Architettura fine aggiornamento del database

In questo schema UML_G , è presente il servizio *EndUpdateService*, che riveste un ruolo fondamentale nella fase conclusiva del $processo_G$ di aggiornamento dei $dati_G$ raccolti dallo $scraper_G$. Dopo che lo $spider_G$, rappresentato da *SpiderVimar*, ha terminato il prelievo delle informazioni dal sito aziendale e che *DBPipeline* ha processato i diversi tipi di $dati_G$ estratti, è necessario chiudere formalmente il ciclo di aggiornamento avviato. In questo contesto entra in gioco *EndUpdateService*.

La procedura inizia quando il *DBPipeline*, al termine del processamento degli item, invoca il metodo *end_update()* esposto dall'interfaccia *EndUpdateUseCase*. Questa interfaccia ha il compito di orchestrare la chiusura, delegando la responsabilità operativa concreta proprio a *EndUpdateService*. Quest'ultimo incapsula la logica necessaria per completare correttamente il $processo_G$ e si appoggia a sua volta all'interfaccia *EndUpdatePort* per comunicare con il livello di infrastruttura.

EndUpdatePort si affida all'implementazione fornita da *SupabaseAdapter*, il quale ha il compito di tradurre l'operazione di chiusura in una chiamata compatibile con il $Repository_G$, rappresentato da *SupabaseRepository*. Questo $Repository_G$, infine, interagisce con il $database_G$ per registrare che il $processo_G$ di aggiornamento si è concluso. L'esito dell'operazione viene incapsulato in un oggetto *DbInsertOperationResponse*, che riporta se l'operazione è andata a buon fine oppure no, insieme a un eventuale messaggio di errore o conferma.



3.5.4 Architettura dell'inserimento associazione prodotto - pdf nel database

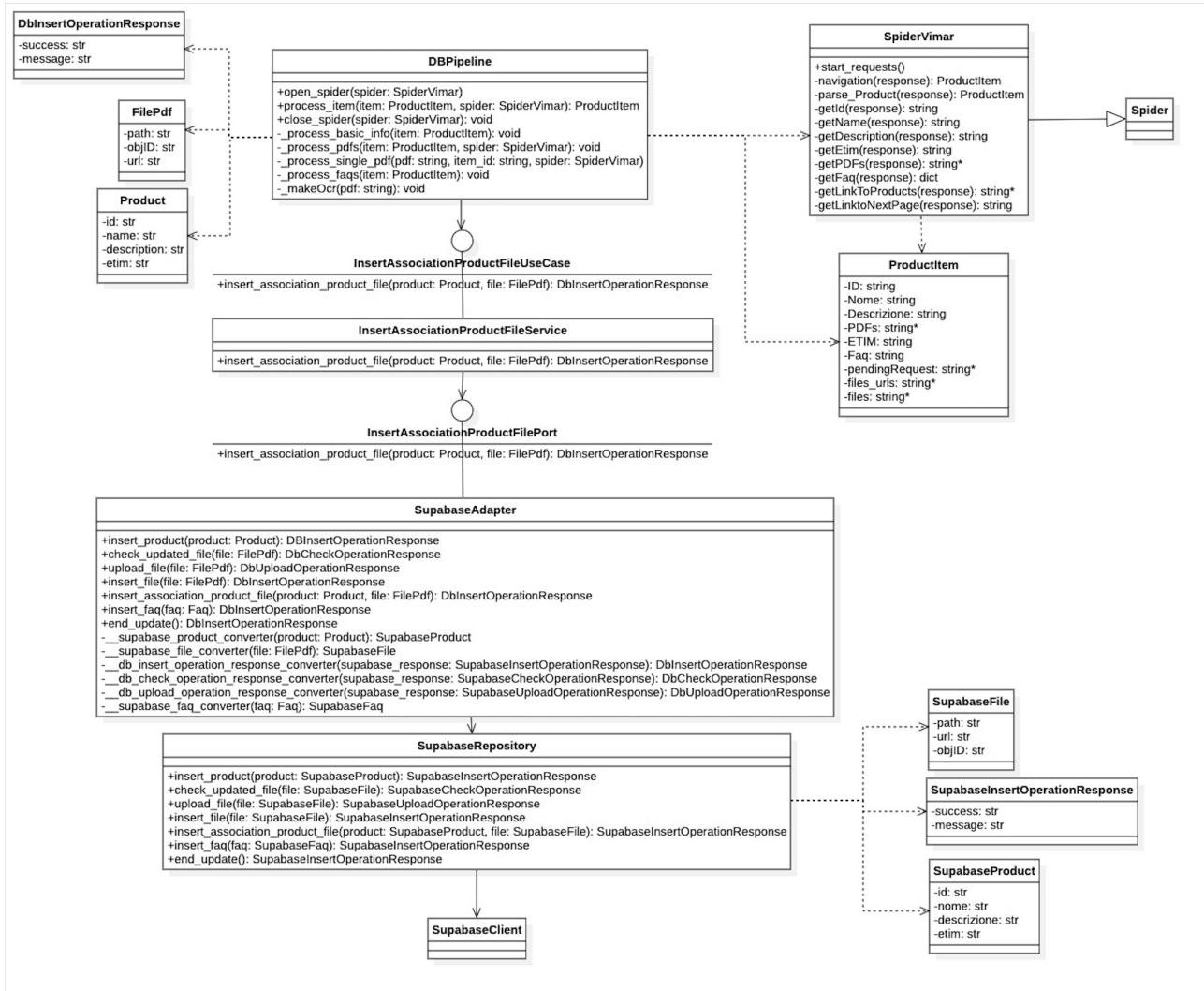


Figure 4: Architettura dell'inserimento associazione prodotto - pdf nel database

Il servizio *InsertAssociationProductFile* rappresenta il cuore operativo per l'associazione tra i prodotti estratti dallo *scraper_G* e i relativi documenti PDF. Questo componente nasce dall'esigenza di garantire un legame strutturato e tracciabile tra entità distinte ma correlate, assicurando al contempo l'integrità dei *dati_G* e la coerenza del *sistema_G*.

Tutto ha inizio quando lo *spider_G* *SpiderVimar*, durante la navigazione del sito aziendale, raccoglie le informazioni fondamentali sui prodotti e individua i PDF collegati. Il *DBPipeline*, vero regista dell'elaborazione, prende in carico questi *dati_G* grezzi e li lavora attraverso una serie di metodi specializzati.

Una volta completata questa fase preparatoria, il *sistema_G* attiva il caso d'uso *InsertAssociationProductFileUseCase*, che funge da intermediario tra la logica operativa dello *scraper_G* e le esigenze del business.

L'implementazione dell'interfaccia messa a disposizione da *InsertAssociationProductFileUseCase* è fornita da *InsertAssociationProductFileService*. Il servizio opera come un filtro rigoroso, assicurandosi che solo le associazioni corrette e complete possano essere memorizzate all'interno del *database_G*.

Per dialogare con l'infrastruttura di persistenza, il servizio si affida all'interfaccia *InsertAssociationProductFilePort*. Nella concretezza operativa, è il *SupabaseAdapter* a realizzare questa interfaccia, traducendo gli oggetti dominio in un linguaggio comprensibile a *Supabase_G*. La conversione



trasforma un generico *Product* nel più specifico *SupabaseProduct*, e allo stesso modo adatta il *FilePdf* al formato *SupabaseFile*.

Infine è *SupabaseRepository*, che concretamente interagisce con il *database_G*, eseguendo l'operazione di associazione vera e propria. L'esito di questa operazione viene imballato in una *SupabaseInsertOperationResponse*.

3.5.5 Architettura dell'inserimento delle FAQ nel database

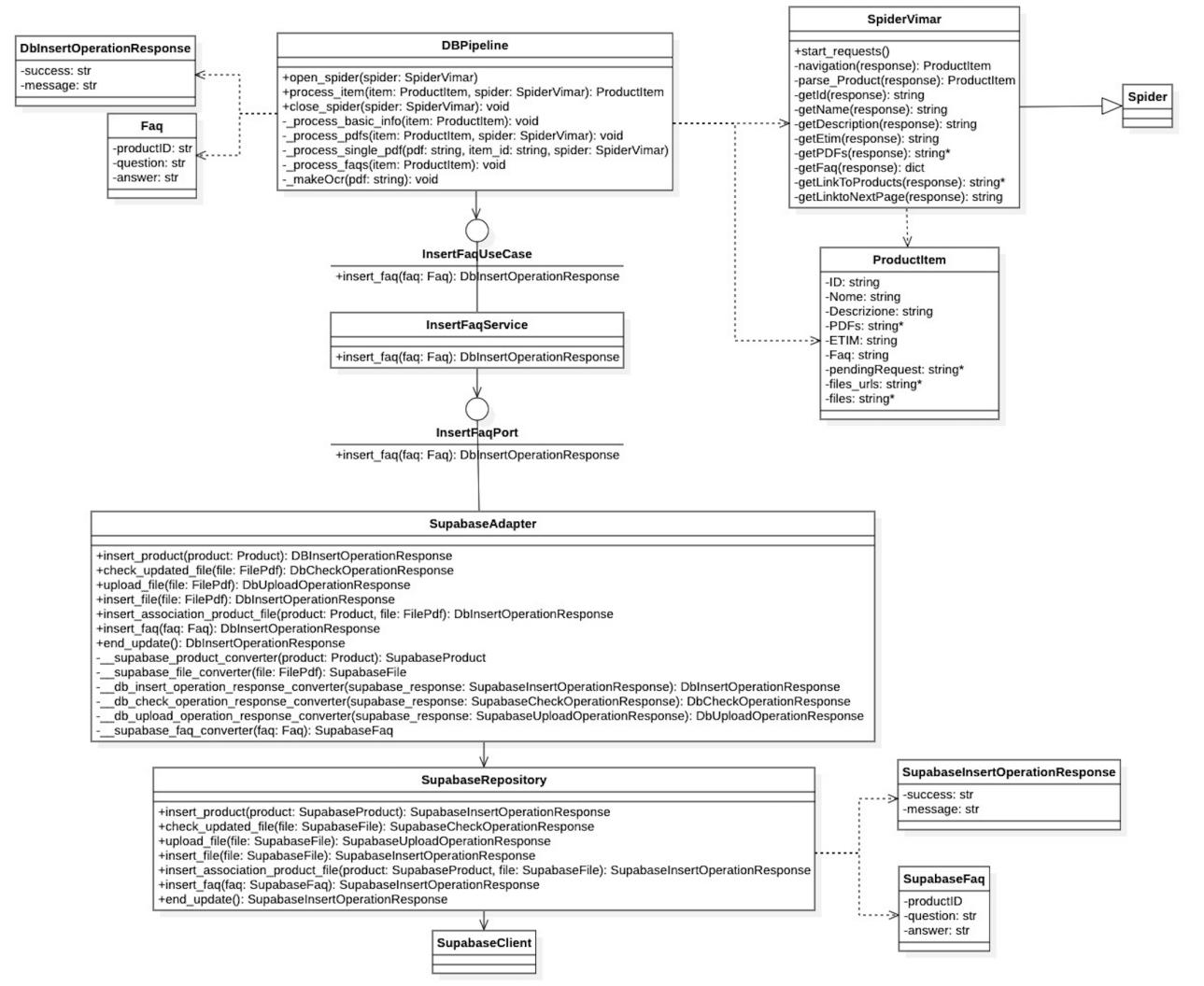


Figure 5: Architettura dell'inserimento delle FAQ nel database

Il servizio *InsertFAQService* rappresenta il meccanismo chiave per gestire l'inserimento *FAQ_G* associate ai prodotti estratti dallo *scraper_G SpiderVimar*.

L'intero *processo_G* inizia quando lo *SpiderVimar*, durante la navigazione del sito aziendale, identifica e raccoglie le informazioni relative alle *FAQ_G* attraverso il metodo *getFAQ()*, che estrae una struttura di dati complessa contenente sia le domande che le risposte associate. Questi *dati_G* grezzi vengono poi passati al *DBPipeline*, che assume il ruolo di coordinatore principale dell'elaborazione. All'interno del pipeline, il metodo *_process_faqs()* si occupa specificamente di elaborare queste informazioni, preparandole per le fasi successive.

Successivamente, il caso d'uso *InsertFAQUseCase*, espone il metodo *insertFAQ()*, il quale viene implementato dal servizio *InsertFAQService*.



Per comunicare con il $database_G$, il servizio si affida all'interfaccia $InsertFaqPort$, un contratto ben definito che garantisce l'indipendenza del $sistema_G$ dalle specifiche implementazioni tecniche. Nella pratica concreta, è il $SupabaseAdapter$ a realizzare questa interfaccia, svolgendo il cruciale lavoro di traduzione tra gli oggetti dominio e il formato specifico richiesto da $Supabase_G$. L' $Adapter_G$ utilizza il metodo $_supabase_faq_converter()$ per trasformare un generico oggetto Faq nella sua controparte $SupabaseFaq$, assicurandosi che tutti i campi (productID, question e answer) vengano mappati correttamente secondo le specifiche del $database_G$.

Infine è $SupabaseRepository$ che materialmente interagisce con il $database_G$ attraverso l'operazione $insert_faq()$, persistendo i $dati_G$ nel formato appropriato. L'esito di questa operazione viene restituito come $SupabaseInsertOperationResponse$, una struttura che contiene il classico flag di successo e un messaggio descrittivo.

3.5.6 Architettura dell'inserimento dei file nel database

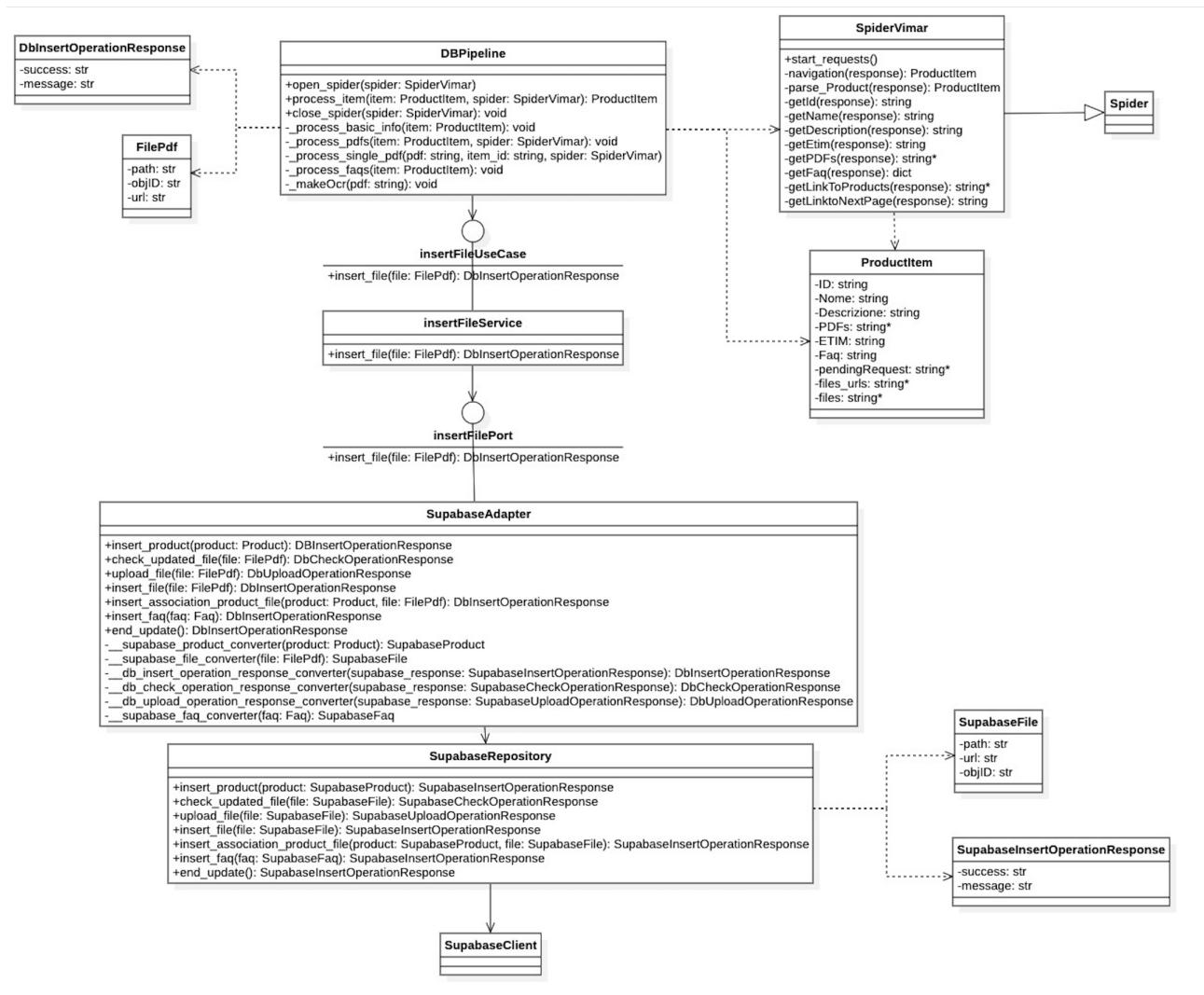


Figure 6: Architettura dell'inserimento dei file nel database

Il servizio $InsertFileService$ costituisce il nucleo operativo per la gestione dell'inserimento dei documenti PDF all'interno del $database_G$. Lo spider G $SpiderVimar$, durante la navigazione del sito web aziendale, identifica e raccoglie i riferimenti ai file PDF attraverso il metodo $getPDFs()$, estraendo informazioni cruciali come il percorso, l'URL e un identificativo univoco per ciascun documento.



Questi $dati_G$ grezzi vengono poi passati al $DBPipeline$, che assume il ruolo di regista principale del $processo_G$ di elaborazione. Il metodo $_process_pdfs()$ coordina la gestione dei documenti, mentre $_process_single_pdf()$ si occupa dell'elaborazione individuale di ciascun file. Un passaggio particolarmente significativo è rappresentato dal metodo $makeOcr()$, che svolge la funzione critica di estrazione del testo dai PDF attraverso tecnologia OCR_G , trasformando così documenti scannerizzati o in formato immagine in contenuti testuali ricercabili e analizzabili. Questa operazione è essenziale per garantire che i documenti siano pienamente integrabili nel $sistema_G$ e utilizzabili per successive elaborazioni.

Successivamente i metodi esposti dall'interfaccia $InsertFileUseCase$ vengono implementati effettivamente dal servizio $InsertFileService$, il quale implementata la logica sostanziale per la validazione, la normalizzazione e la preparazione finale dei $dati_G$ prima della persistenza.

Per comunicare con il livello di infrastruttura, il servizio utilizza l'interfaccia $InsertFilePort$, che definisce un contratto chiaro e indipendente dalle implementazioni concrete. Nella pratica, è il $SupabaseAdapter$ a realizzare questa interfaccia, svolgendo il cruciale lavoro di traduzione tra gli oggetti dominio e il formato specifico richiesto da $Supabase_G$. L' $Adapter_G$ impiega il metodo $_supabase_file_converter()$ per trasformare un oggetto $FilePdf$ nella sua controparte $SupabaseFile$, assicurando che tutti i campi (path, url e objID) vengano mappati correttamente secondo le specifiche del $database_G$.

L'operazione concreta di inserimento viene eseguita dal $SupabaseRepository$ attraverso il metodo $insert_file()$, che persiste i $dati_G$ nel $database_G$. L'esito dell'operazione viene restituito come $SupabaseInsertOperationResponse$, che include un flag di successo e un messaggio descrittivo.



3.5.7 Architettura dell'inserimento dei prodotti nel database

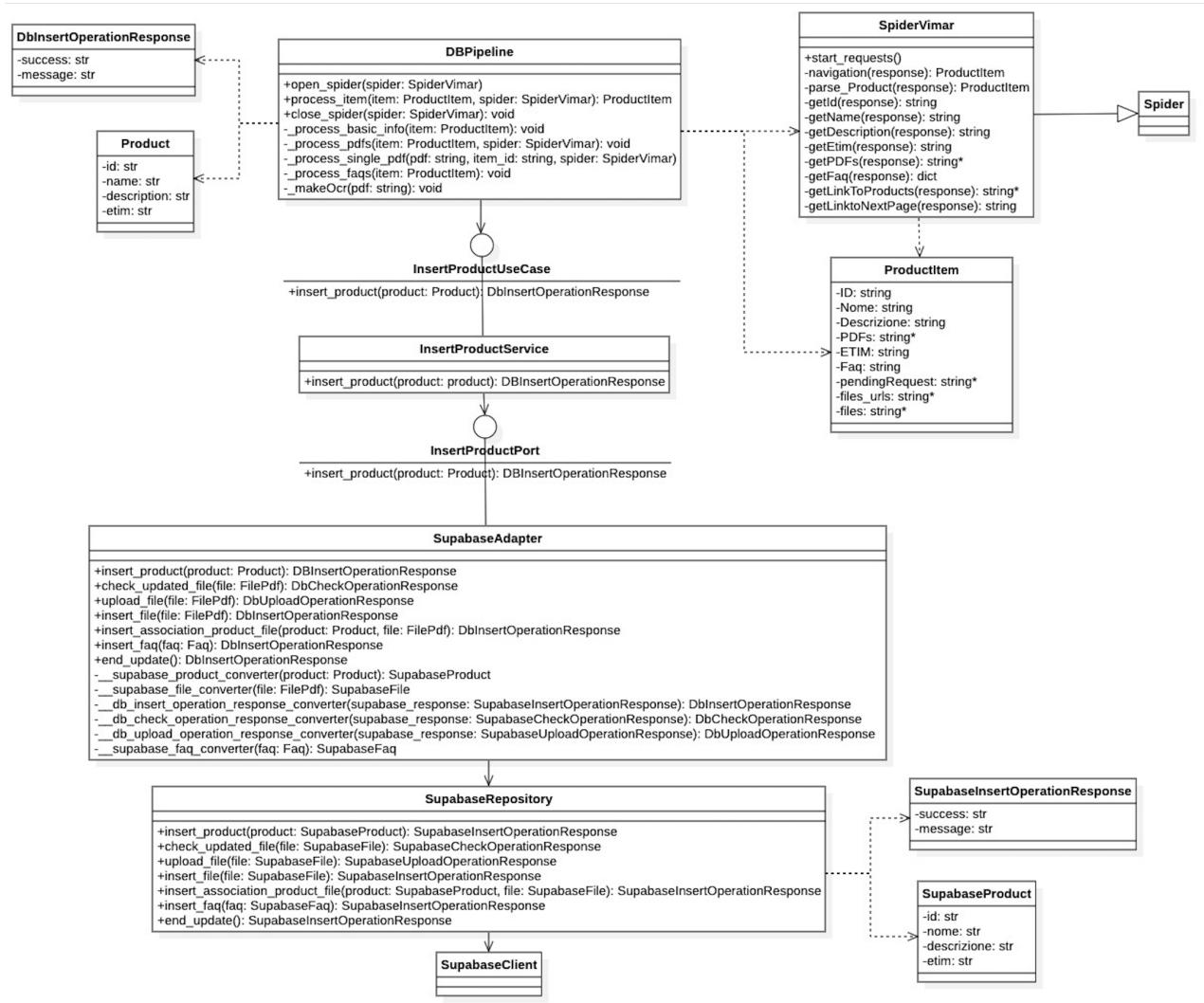


Figure 7: Architettura dell'inserimento dei prodotti nel database

Il servizio *Insert ProductService* costituisce il componente fondamentale per l'inserimento dei prodotti estratti dallo *dati_G* *SpiderVimar* all'interno del *database_G*.

L'intero flusso prende avvio quando lo *database_G* *SpiderVimar* identifica e raccoglie le informazioni base dei prodotti attraverso i metodi `getId()`, `getName()`, `getDescription()` e `getEthn()`. Questi *dati_G* grezzi vengono poi passati al *DBPipeline*, che coordina il *processo_G* di elaborazione attraverso una serie di metodi specializzati. In particolare, il metodo `_process_basic_Info()` si occupa di organizzare e strutturare le informazioni fondamentali del prodotto.

Il caso d'uso *InsertProductUseCase* riceve i prodotti così preparati e avvia la procedura di inserimento vero e proprio. La logica operativa è incarnata dal *Insert ProductService*, che rappresenta il cuore del *processo_G*. Questo servizio riceve un oggetto *Product* completo di tutti i campi obbligatori (*id*, *name*, *description*, *ethn*, verifica la validità e completezza dei *dati_G*, prepara l'oggetto per la persistenza e infine delega l'operazione concreta al *Port_G* implementato.

L'interfaccia *InsertProductPort* definisce il contratto per la persistenza, implementato concretamente dal *SupabaseAdapter*. Questo componente svolge un ruolo cruciale nella conversione degli oggetti dominio nel formato specifico richiesto da *Supabase_G*. Attraverso il metodo `_supabase_product_converter()`, trasforma un generico oggetto *Product* nella sua controparte *SupabaseProduct*, garantendo che tutti i



campi vengano mappati correttamente secondo le specifiche del $database_G$.

L'operazione di inserimento materiale viene eseguita dal *SupabaseRepository* attraverso il metodo *insert_product()*, che interagisce direttamente con il $database_G$. L'esito dell'operazione viene restituito come *SupabaseInsertOperationResponse* che include un flag di successo e un messaggio descrittivo dell'esito.

3.5.8 Architettura dell'upload dei file nel database

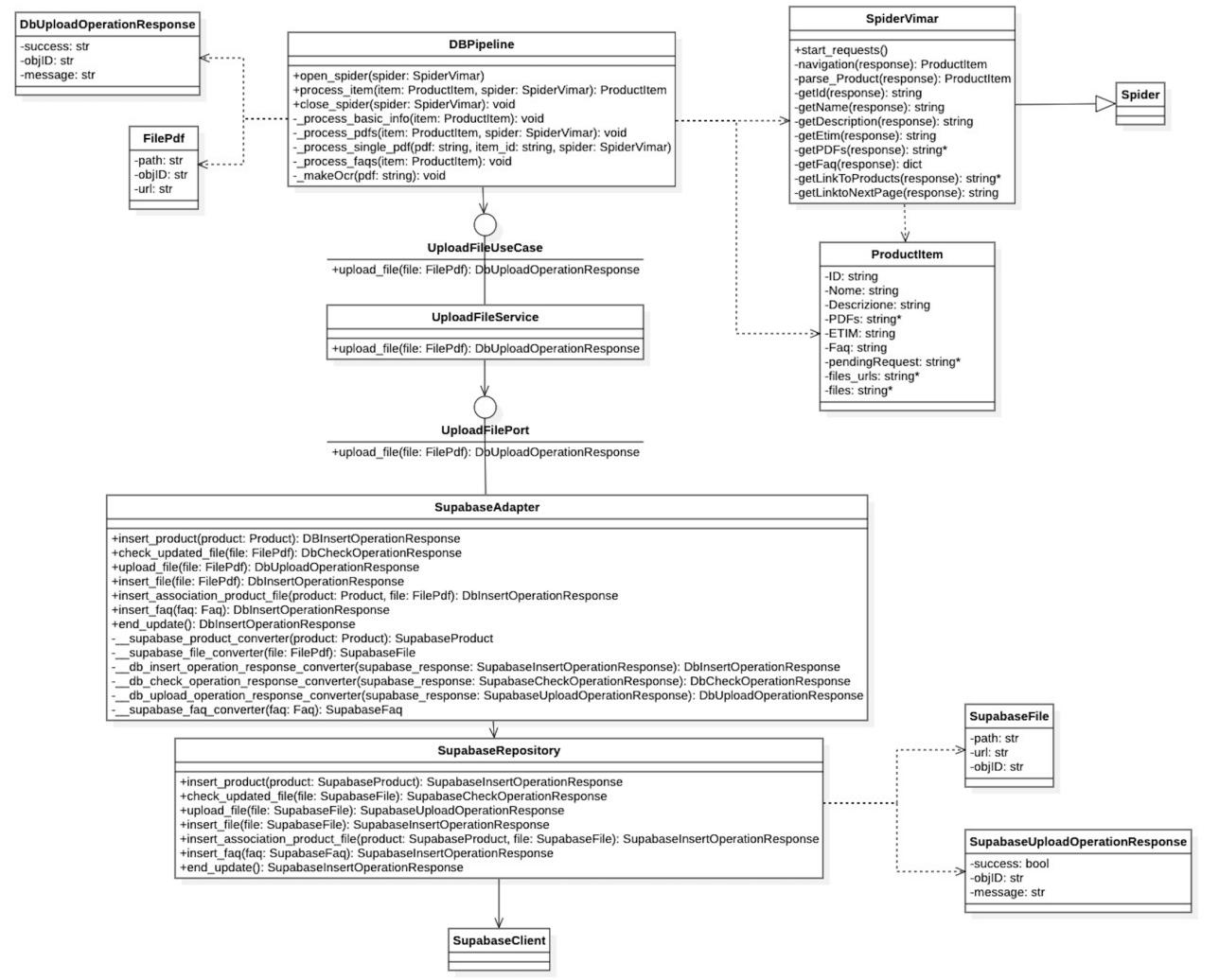


Figure 8: Architettura dell'upload dei file nel database

Il servizio *UploadFileService* rappresenta il componente specializzato nella gestione del caricamento dei documenti PDF all'interno dell'infrastruttura di storage.

Lo *spider_G* *SpiderVimar*, durante la navigazione del sito target, identifica i documenti PDF disponibili attraverso il metodo *getPDFs()*. Questi file, insieme ai relativi metadati, vengono passati al *DBPipeline* per l'elaborazione iniziale. All'interno del pipeline, il metodo *_process_single_pdf()* si occupa dell'elaborazione individuale di ciascun file.

Il caso d'uso *UploadFileUseCase* riceve i file così preparati e avvia la procedura di caricamento vero e proprio. La logica operativa è incarnata dal *UploadFileService*, che implementa i metodi esposti da *UploadFileUseCase*. Questo servizio implementa una serie di controlli essenziali sui *dati_G* estratti.

L'interfaccia *UploadFilePort* definisce il contratto per l'interazione con il *sistema_G* di storage, implementato concretamente dal *SupabaseAdapter*. Questo componente svolge un ruolo cruciale nella traduzione



degli oggetti dominio nel formato specifico richiesto da *Supabase_G*. Attraverso il metodo *_supabase_file_converter()*, trasforma un generico oggetto *FilePdf* nella sua controparte *SupabaseFile*, assicurando che tutti i campi (path, url e objID) vengano mappati correttamente secondo le specifiche dell'*API_G* di *Supabase_G* Storage.

L'operazione di caricamento effettiva viene eseguita dal *SupabaseRepository* attraverso il metodo *upload_file()*, che interagisce direttamente con le *API_G* di storage di *Supabase_G*. La risposta dell'operazione, di tipo *SupabaseUploadOperationResponse*, viene poi convertita dall'*Adapter_G* in una *DbUploadOperationResponse* standardizzata, che include tre informazioni fondamentali:

- *success*: flag booleano che indica l'esito dell'operazione;
- *objID*: l'identificativo univoco assegnato al file nello storage;
- *message*: descrizione testuale dell'esito, particolarmente utile in caso di errori.

3.6 Struttura del sistema

3.6.1 Frontend

Lo sviluppo del *frontend_G* è stato guidato da un approccio flessibile, che integra in modo coerente le pratiche consolidate del framework *Angular_G*, senza però conformarsi in maniera rigida a un singolo modello architettonale, come *MVC_G* o *MVVM_G*. Questa strategia ha permesso di modellare l'*architettura_G* dell'interfaccia *utente_G* in modo dinamico e adattabile, rispondendo efficacemente alle esigenze specifiche del *progetto_G*.

L'adozione di una struttura architettonale non vincolata ha reso possibile il raggiungimento di un equilibrio ottimale tra modularità, scalabilità del codice e semplicità nella gestione dello stato, favorendo al contempo la manutenibilità e l'evolutività dell'*applicazione_G* nel lungo termine.

La struttura dell'*applicazione_G* è organizzata in due aree funzionali distinte, corrispondenti alle esigenze dell'*utente_G* finale e dell'amministratore del *sistema_G*.

Per quanto riguarda l'area *utente_G*, essa include componenti come *SidebarComponent*, *NavigatorComponent*, *FooterComponent* e il *ChatComponent*, i quali definiscono l'interfaccia e l'esperienza di utilizzo dell'*utente_G* finale. L'area amministrativa, invece, è costituita da componenti quali *DashboardPageComponent* e *LoginPageComponent*, destinati alla gestione e al monitoraggio del *sistema_G* da parte degli operatori autorizzati.

I componenti sono responsabili della presentazione, mentre i servizi sono responsabili dell'accesso ai dati. La comunicazione con il *backend_G*, quindi, avviene attraverso richieste inviate dai servizi, che a loro volta vengono instradate attraverso un *API Gateway_G* basato su *Kong_G*, che agisce da punto di ingresso centralizzato, consentendo il corretto reindirizzamento delle chiamate verso i microservizi esposti da *Supabase_G*. Questo livello di astrazione favorisce un'*architettura_G* più sicura e scalabile.

Questo approccio consente di delegare ai servizi applicativi operazioni fondamentali come la creazione di nuove conversazioni, l'invio e la ricezione di messaggi, la gestione dei feedback utente, nonché il login dell'amministratore. In tal modo, i componenti dell'interfaccia si mantengono snelli, focalizzati esclusivamente sulla presentazione, e quindi più facilmente riutilizzabili, testabili e manutenibili.

La gestione dello stato rappresenta un elemento centrale nell'*architettura_G* dell'*applicazione_G*. A tal fine, sono stati introdotti dei servizi dedicati, *ConversationService* e *UserService*, che adottano l' *Observer pattern_G* tramite l'uso di *BehaviorSubject_G* forniti da *RxJS_G*. Questi servizi fungono da punto di aggregazione per lo stato delle conversazioni, gestendo in maniera centralizzata quest'ultime, i messaggi e la conversazione attiva per ciascun *utente_G*. Attraverso l'esposizione di flussi observable, i componenti possono sottoscriversi ai cambiamenti di stato e aggiornare dinamicamente la propria interfaccia in risposta a cambiamenti. Questo approccio garantisce un flusso di dati reattivo, riducendo notevolmente il *coupling_G* tra componenti.



Un aspetto molto interessante è l'utilizzo dell' *anonymous sing-in*, offerto da *Supabase_G*, che permette lo sviluppo di un'applicazione_G in grado di offrire agli utenti_G un'esperienza autenticata senza la necessità di creare un account_G con delle credenziali dedicate. L'applicativo_G, inoltre, presenta un'area riservata agli amministratori accessibile attraverso l'inserimento di credenziali. È possibile utilizzare le politiche di sicurezza a livello di riga (*Row-Level Security, RLS*) per distinguere tra un utente anonimo e uno permanente, verificando la presenza del claim *is_anonymous* all'interno del JWT_G restituito dalla funzione *auth.jwt()*.

3.6.1.1 Componenti principali

ChatComponent

ChatComponent gestisce l'interfaccia utente e la logica per la funzionalità di chat.

I suoi metodi principali sono:

- *newChat()*: si occupa di creare una nuova conversazione per l'*utente_G* corrente;
- *handleFeedbackClick()*: si occupa di gestire il *feedback_G* dell'*utente_G* su un messaggio specifico. Questo metodo apre un dialogo per raccogliere il *feedback_G* e lo invia al *database_G* tramite il servizio *SupabaseService*;
- *startTimeout()* e *clearTimeout()*: si occupano della gestione del timeout per il messaggio corrente. Questi metodi vengono utilizzati per garantire che l'applicazione_G restituisca una risposta entro un range di tempo limitato. Se la risposta non dovesse essere elaborata e visualizzata entro tale range, l'applicativo_G chiederà all'*utente_G* la riemissione della domanda.

MessageboxComponent

MessageboxComponent gestisce l'input dell'*utente_G* per inviare messaggi in una conversazione. Si occupa di raccogliere il messaggio, validarlo, inviarlo al server e aggiornare l'interfaccia utente in tempo reale.

I suoi metodi principali sono:

- *handleSendMessage()*: si occupa di gestire l'invio di un messaggio da parte dell'*utente_G*. Questo include la validazione del messaggio, la creazione di un messaggio temporaneo nell'interfaccia utente, l'invio del messaggio al server tramite il servizio *SupabaseService* e l'aggiornamento del messaggio temporaneo con l'ID definitivo una volta ricevuta la risposta dal server;
- *handleInputChange()*: si occupa di gestire i cambiamenti nell'input del messaggio da parte dell'*utente_G*. Questo metodo aggiorna il messaggio corrente, calcola i caratteri rimanenti e verifica se il messaggio contiene parole vietate;
- *handleKeyDown()*: si occupa di gestire l'evento di pressione di un tasto all'interno del campo di input del messaggio. In particolare, verifica se l'*utente_G* ha premuto il tasto "Enter" e, in tal caso, chiama il metodo *handleSendMessage()* per inviare il messaggio;
- *startTimeout()* e *clearTimeout()*: si occupano della gestione del timeout per il messaggio corrente. Questi metodi vengono utilizzati per garantire che l'

SidebarComponent

SidebarComponent gestisce la barra laterale dell'applicazione_G. La sua funzione principale è fornire un'interfaccia per navigare tra le diverse conversazioni e interagire con alcune funzionalità dell'applicazione_G, come la creazione, la selezione e l'eliminazione delle chat. Inoltre mette a disposizione degli amministratori la funzionalità di navigazione tra la *home page* e la *Dashboard*.

I suoi metodi principali sono:

- *toggleChat()*: si occupa di gestire l'apertura e la chiusura della barra laterale (sidebar) dell'applicazione_G. Questo metodo viene tipicamente utilizzato per migliorare l'esperienza utente su



dispositivi con schermi più piccoli o per nascondere/mostrare la barra laterale in base alle interazioni dell'utente;

- *handleClickOutsideEvent()*: si occupa di gestire il comportamento della sidebar quando l'*utente_G* clicca al di fuori di essa;
- *newChat()*: si occupa di creare una nuova conversazione per l'*utente_G* corrente;
- *selectChat()*: si occupa di gestire la selezione di una chat specifica da parte dell'*utente_G*. Quando l'*utente_G* seleziona una chat dall'elenco, questo metodo aggiorna lo stato della conversazione corrente e notifica il servizio *ConversationService*;
- *deleteChat()*: si occupa di eliminare una chat specifica dall'elenco delle conversazioni dell'utente. Dopo l'eliminazione, aggiorna l'elenco delle chat e gestisce lo stato della conversazione corrente;
- *updateLinkLabel()*: si occupa di aggiornare dinamicamente l'etichetta del link di navigazione tra la *Dashboard* e la *home page* in base all'URL corrente;
- *navigateToDashOrHome()*: si occupa di gestire la navigazione tra la *Dashboard* e la *home page* in base all'URL corrente. Inoltre, chiude la sidebar e aggiorna lo stato visivo del pulsante "burger" dopo la navigazione.

FooterComponent

FooterComponent gestisce il footer dell'*applicazione_G*. La sua funzione principale è fornire un'interfaccia per la navigazione tra la *home page* e la pagina di Login, oltre a gestire lo stato di autenticazione dell'*utente_G* e la funzionalità di logout.

I suoi metodi principali sono:

- *updateLinkLabel()*: si occupa di aggiornare dinamicamente l'etichetta del link di navigazione nel footer in base all'URL corrente. Questo permette di fornire un'indicazione chiara all'*utente_G* su quale pagina verrà aperta cliccando sul link;
- *navigateToLoginOrHome()*: si occupa di gestire la navigazione tra la *home page* e la pagina di Login in base all'URL corrente. Inoltre, deseleziona eventuali chat selezionate nell'interfaccia utente;
- *showLogout()*: si occupa di mostrare o nascondere una finestra di conferma per il logout. Questo metodo utilizza la manipolazione del *DOM_G* per aggiungere o rimuovere una classe CSS che controlla la visibilità della finestra di conferma;
- *onLogout()*: si occupa di gestire il processo di logout dell'*utente_G*. Questo include la disconnessione dell'utente autenticato, la rimozione del *token_G* di sessione e il passaggio a una sessione anonima. Inoltre, aggiorna lo stato di autenticazione e ricarica la pagina per riflettere i cambiamenti.

3.6.1.2 Servizi

SupabaseService

SupabaseService funge da interfaccia tra l'*applicazione_G* e il *backend_G* di *Supabase_G*. Come già detto, *Supabase_G* è una piattaforma *backend_G* che fornisce funzionalità come autenticazione, *database PostgreSQL_G*, storage e funzioni serverless. Questo servizio è progettato per gestire tutte le operazioni relative a *Supabase_G*, come autenticazione, gestione delle chat, invio di messaggi e analisi dei *dati_G*.

I suoi metodi principali sono:

- *signInAnonymously()*: si occupa di effettuare il login anonimo di un *utente_G* utilizzando le funzionalità di autenticazione di *Supabase_G*. Questo metodo consente di creare una sessione temporanea per un utente che non ha effettuato il login con credenziali (email e password) o altri metodi di autenticazione;



- *signInWithPassword()*: si occupa di gestire il login di un utente utilizzando le credenziali di email e password. Questo metodo utilizza le funzionalità di autenticazione di *Supabase_G* per verificare le credenziali dell'*utente_G* e creare una sessione autenticata;
- *saveAnonSession()*: si occupa di salvare la sessione anonima di un *utente_G* nei cookie;
- *checkAnonSession()*: si occupa di verifica se esiste una sessione anonima salvata nei cookie;
- *getUser()*: si occupa di recuperare le informazioni sull'utente attualmente autenticato tramite *Supabase_G*. Questo metodo è utile per ottenere dettagli come l'ID dell'*utente_G*, l'email e altre informazioni associate alla sessione corrente;
- *getSession()*: si occupa di recuperare la sessione attualmente attiva dell'*utente_G* tramite *Supabase_G*;
- *setSession()*: si occupa di impostare una sessione specifica per l'*utente_G* utilizzando le funzionalità di autenticazione di *Supabase_G*;
- *signOut()*: si occupa di effettuare il logout dell'*utente_G* attualmente autenticato. Questo metodo utilizza le funzionalità di autenticazione di *Supabase_G* per terminare la sessione dell'utente e rimuovere eventuali *token_G* di autenticazione;
- *getChatsForUser()*: si occupa di recuperare tutte le chat associate a un determinato *utente_G* dal database *PostgreSQL_G* di *Supabase_G*. Questo metodo aggiorna un *BehaviorSubject_G* per mantenere sincronizzato lo stato delle chat nell'*applicazione_G*;
- *deleteChat()*: si occupa di eliminare una chat specifica dal *database_G*. Dopo l'eliminazione, aggiorna l'elenco delle chat mantenuto localmente tramite un *BehaviorSubject_G*;
- *handleSupabaseError()*: si occupa di gestire gli errori di comunicazione con *Supabase_G*;
- *testSupabaseConnection()*: si occupa di testare la connessione a *Supabase_G* effettuando una semplice query sulla tabella "chat";
- *newChat()*: si occupa di creare una nuova chat per un *utente_G* specifico nel *database_G*. Dopo aver creato la chat, aggiorna l'elenco delle chat tramite un *BehaviorSubject_G*;
- *sendQuestion()*: si occupa di inviare un messaggio (domanda) a una chat specifica nel *database_G*. Questo metodo inserisce un nuovo record nella tabella "messaggio" e restituisce i *dati_G* del messaggio appena creato;
- *submitFeedback()*: si occupa di aggiornare un messaggio nel *database_G* con il *feedback_G* fornito dall'*utente_G*. Questo *feedback_G* può essere positivo o negativo e include un testo opzionale che descrive il *feedback_G*;
- *listenMessageChanges()*: si occupa di ascoltare i cambiamenti in tempo reale su una riga specifica della tabella "messaggio" nel *database_G*. Questo metodo utilizza i canali di *Supabase_G* per ricevere notifiche quando una riga viene aggiornata, consentendo all'*applicazione_G* di reagire immediatamente ai cambiamenti;
- *getCountFeedbackMex()*: si occupa di recuperare il numero di messaggi relativi ai *feedback_G* positivi, negativi e neutri dal *database_G*. Questo metodo utilizza una funzione *RPC_G* (Remote Procedure Call) definita nel *database_G* per eseguire una query personalizzata;
- *getCountMessages()*: si occupa di recuperare il numero di messaggi inviati, ordinati per settimana in ordine crescente (dal più vecchio al più recente). Anche questo metodo utilizza una funzione *RPC_G*;
- *getCountFeedbacks()*: si occupa di recuperare il numero di *feedback_G* positivi e negativi per settimana dal *database_G* attraverso una funzione *RPC_G*;



- *getAnalyzeTextMessages()*: si occupa di analizzare i messaggi inviati dagli $utenti_G$ per calcolare statistiche come il numero medio di parole per messaggio e i termini più utilizzati. Questo metodo utilizza una *edge function_G* di *Supabase_G* per eseguire l'analisi.

ConversationService

ConversationService gestisce lo stato delle conversazioni e dei messaggi nell'*applicazione_G*. Fornisce un'interfaccia centralizzata per aggiornare, monitorare e sincronizzare le conversazioni e i messaggi tra i componenti dell'*applicazione_G* e il *backend_G*.

I suoi metodi principali sono:

- *setCurrentConversation()*: si occupa di impostare la conversazione corrente selezionata dall'*utente_G*. Questo metodo aggiorna lo stato globale della conversazione corrente e notifica tutti i componenti sottoscritti tramite un *BehaviorSubject_G*;
- *addTempMessage()*: si occupa di aggiungere un messaggio temporaneo alla conversazione corrente. Questo è utile per mostrare immediatamente il messaggio nell'interfaccia utente prima che venga salvato nel *backend_G*;
- *updateTempMessage()*: si occupa di aggiornare un messaggio temporaneo con i *dati_G* definitivi ricevuti dal *backend_G*. Questo metodo è utile per sostituire un messaggio temporaneo con il messaggio definitivo che include l'ID reale restituito dal *backend_G*;
- *updateMessage()*: si occupa di aggiornare un messaggio esistente con nuovi *dati_G*. Questo metodo viene utilizzato per sincronizzare i messaggi con il *backend_G* o per aggiornare i messaggi nell'interfaccia utente in tempo reale.

UserService

UserService gestisce lo stato dell'*utente_G* nell'*applicazione_G*. Fornisce un'interfaccia per inizializzare, monitorare e accedere ai *dati_G* dell'*utente_G*, sia che si tratti di un utente autenticato che di un utente anonimo. Questo servizio utilizza il *SupabaseService* per interagire con il *backend_G* e gestire l'autenticazione.

I suoi metodi principali sono:

- *initUser()*: si occupa di inizializzare lo stato dell'utente nell'*applicazione_G*. Questo metodo verifica se esiste un utente autenticato tramite *Supabase_G* e, in caso contrario, crea una sessione anonima per l'*utente_G*;
- *getCurrentUser()*: si occupa di restituire l'*utente_G* attualmente memorizzato nel *BehaviorSubject_G* "userSubject". Questo metodo fornisce un modo sincrono per accedere ai *dati_G* dell'utente corrente senza dover sottoscriversi all'osservabile "user".

ChartService

ChartService gestisce la creazione e il rendering di grafici utilizzando la libreria *Chart.js_G*. Questo servizio fornisce un'interfaccia flessibile per generare diverse tipologie di grafici e supporta l'uso di strategie per personalizzare il comportamento dei grafici.

I suoi metodi principali sono:

- *setStrategy()*: si occupa di impostare una strategia di rendering per i grafici. Questo metodo utilizza il *pattern Strategy*, che consente di definire diverse modalità di rendering dei grafici e di applicarle dinamicamente in base alle esigenze;
- *renderChart()*: si occupa di eseguire il rendering di un grafico utilizzando una strategia di rendering impostata tramite il metodo *setStrategy()*;



- *single()*: si occupa di creare un grafico con un singolo dataset utilizzando la libreria *Chart.jsG*;
 - *double()*: si occupa di creare un grafico con due dataset utilizzando la libreria *Chart.jsG*;
 - *pieChartData()*: si occupa di creare un grafico a torta utilizzando la libreria *Chart.jsG*.

3.6.1.3 Area amministratore

LoginPageComponent

LoginPageComponent è un componente che gestisce la pagina di login dell'applicazione_G. Si occupa di autenticare l'utente_G tramite email e password utilizzando il servizio *Supabase_G*, di gestire eventuali errori di login e di reindirizzare l'utente_G alla *Home* se il login ha successo o se l'utente_G è già autenticato.

I suoi metodi principali sono:

- `handleLogin()`: si occupa di gestire il processo di login dell' $utente_G$. Questo metodo autentica l' $utente_G$ utilizzando il servizio `SupabaseService`, gestisce eventuali errori e reindirizza l' $utente_G$ alla *home page* in caso di successo;
 - `navigateToHome()`: si occupa di reindirizzare l' $utente_G$ alla *Home* dell' $applicazione_G$. Questo metodo utilizza il servizio di routing di `Angular_G` per la navigazione.

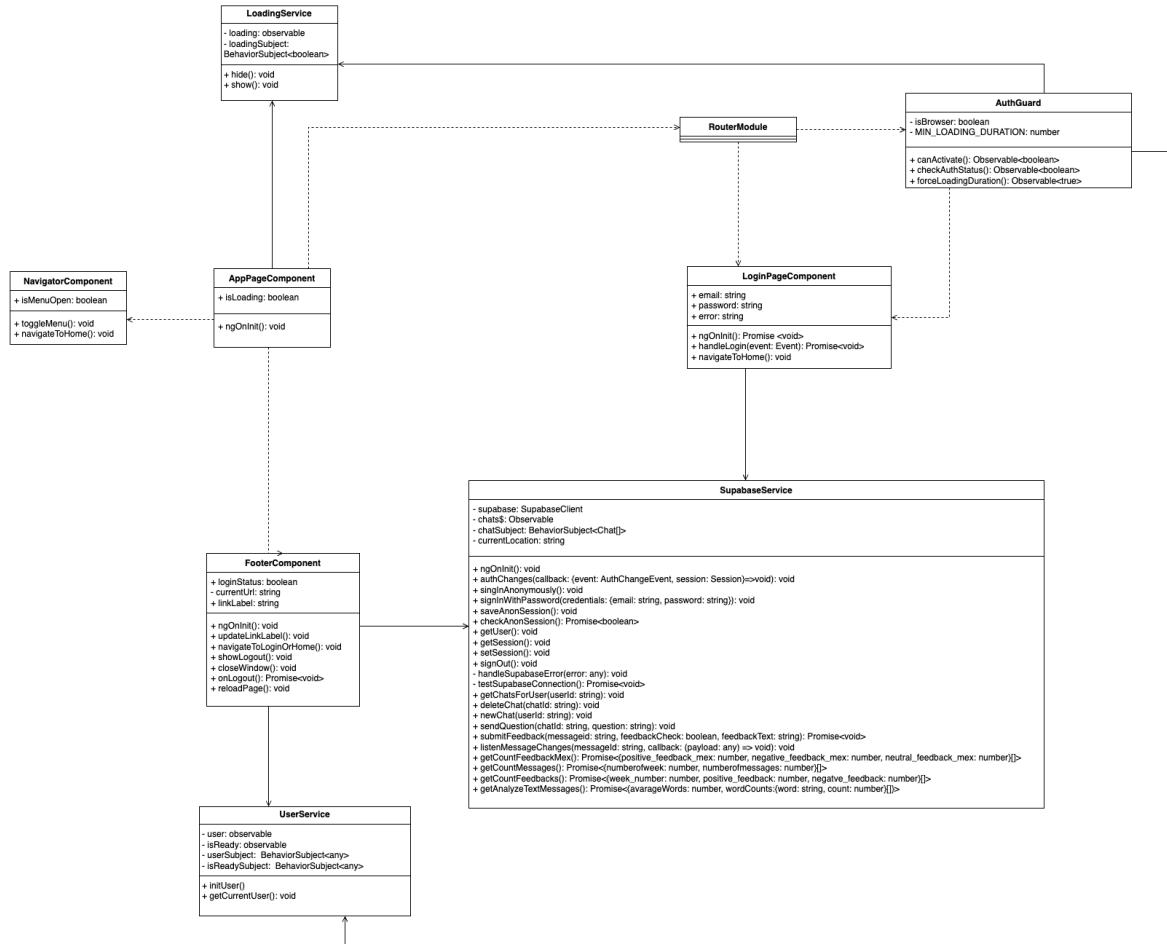


Figure 9: Diagramma delle classi della LoginPageComponent

DashboardPageComponent



DashboardPageComponent è un componente che funge da *dashboard_G* per visualizzare dati analitici e statistiche dell'*applicazione_G*. Utilizza grafici e analisi testuali per fornire una panoramica visiva e dettagliata delle informazioni raccolte.

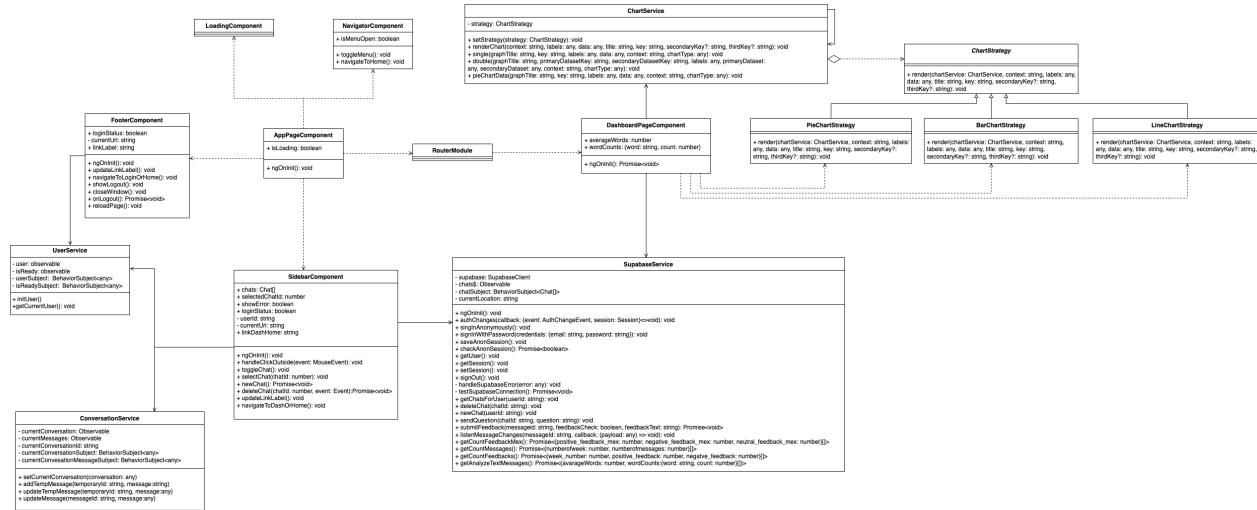


Figure 10: Diagramma delle classi della DashboardPageComponent

ScrapingButtonComponent

ScrapingButtonComponent è un componente che serve per avviare, monitorare e gestire lo stato del *processo_G* di scraping per l'estrazione delle informazioni e dei manuali dei prodotti dal sito web aziendale. I suoi metodi principali sono:

- *startScraping()*: si occupa di avviare il *processo_G* di scraping su un server remoto e di preparare il componente per monitorare lo stato del *processo_G*;
- *startStatusCheck()*: si occupa di avviare un controllo periodico dello stato del *processo_G* di scraping;
- *checkScrapingStatus()*: si occupa di controllare lo stato corrente del *processo_G* di scraping avviato in precedenza inviando una richiesta HTTP GET al server per ottenere informazioni;
- *stopStatusCheck()*: si occupa di interrompere il monitoraggio periodico dello stato del *processo_G* di scraping e di aggiornare lo stato del componente.

AuthGuard

AuthGuard definisce una guardia di route in *Angular_G*, chiamata *AuthGuard*, che protegge le route dell'*applicazione_G* verificando lo stato di autenticazione dell'*utente_G*. Questa guardia impedisce l'accesso a determinate pagine se l'*utente_G* non è autenticato e lo reindirizza alla pagina di login

I suoi metodi principali sono:

- *canActivate()*: è il cuore della guardia di route *AuthGuard*. Questo metodo viene chiamato automaticamente da *Angular_G* quando un *utente_G* tenta di accedere a una route protetta. La sua funzione principale è determinare se l'*utente_G* ha il permesso di accedere alla route;
- *checkAuthStatus()*: si occupa di verificare lo stato di autenticazione dell'*utente_G*. Questo metodo utilizza il servizio *UserService* per determinare se l'*utente_G* è autenticato e non anonimo. Restituisce un *Observable<boolean>* che indica se l'*utente_G* è autorizzato ad accedere alla route protetta;



- *forceLoadingDuration()*: si occupa di garantire che la schermata di caricamento venga mostrata per un tempo minimo definito dalla costante *MIN_LOADING_DURATION*.

3.6.1.4 Design pattern frontend

L'architettura_G del *frontend*_G è stata strutturata seguendo precisi pattern di progettazione software, scelti in funzione delle necessità del *progetto*_G e in conformità con le linee guida consigliate da *Angular*_G.

Di seguito vengono riportati tutti i pattern architetturali utilizzati suddivisi per tipologia.

Pattern creazionali

- **Singleton pattern:** l'intero sistema di *dependency injection*_G di *Angular*_G adotta implicitamente questo pattern architettonico. Tutti i servizi(*SupabaseService*, *ConversationService*, *UserService* e *ChartService*) sono definiti utilizzando il decoratore *@Injectable* con l'opzione *providedIn*: 'root', il che assicura la creazione di un'unica istanza condivisa a livello globale all'interno dell'applicazione_G. Questo meccanismo consente la centralizzazione dello stato applicativo, riduce il rischio di incoerenze e favorisce un uso più efficiente delle risorse attraverso il riutilizzo delle istanze di servizio.

Pattern comportamentali

- **Observer pattern:** questo pattern è implementato attraverso la libreria *RxJS*_G (Reactive Extensions for JavaScript) e viene utilizzato principalmente per gestire eventi asincroni e flussi di dati reattivi.
- **Strategy pattern:** viene utilizzato per gestire in modo flessibile e modulare la creazione di diverse tipologie di grafico utilizzando la libreria *Chart.js*_G. Questo pattern consente di definire una famiglia di algoritmi intercambiabili e di selezionarne uno a runtime senza modificare il codice del contesto che li utilizza.

3.6.2 Backend

Il *sistema*_G è progettato seguendo il modello architettonale esagonale, noto anche come *Ports and Adapters*. La struttura di tale *architettura*_G è rappresentata attraverso un esagono con il *Core*_G (logica di business) al centro e le dipendenze esterne collegate tramite *Port*_G e *Adapter*_G, i quali definiscono uno strato di astrazione in grado di proteggere e isolare il *Core*_G.

Più in dettaglio:

- Il *Controller*_G contiene l'*application logic*_G del *sistema*_G e si occupa della gestione delle richieste in ingresso. In particolare, ha il compito di validare i *dati*_G di tipo *DTO*_G ricevuti in input e convertirli in strutture dati compatibili con il dominio applicativo. Successivamente, il *Controller* invoca lo *Use Case* appropriato per l'esecuzione della *business logic*_G. Al termine dell'elaborazione, adatta la risposta restituita in un oggetto *DTO* e la inoltra al client;
- Il *Core*_G racchiude la *business logic*_G e i casi d'uso dell'applicazione_G. Esso è indipendente da tutte le dipendenze esterne.

Il *Core*_G è composto da:

- *UseCase*_G: rappresenta un caso d'uso specifico del *sistema*_G, che viene implementato da un *service*_G per realizzare la logica di business;
- *Service*_G: contiene la logica di business del *sistema*_G; svolge operazioni specifiche legate al dominio applicativo e delega le interazioni con le altre componenti ai *Port*_G. I *service*_G operano esclusivamente su tipi di *dato*_G di business, assicurando così l'indipendenza della logica di business dal resto dell'architettura_G;
- *Models*_G: contengono le entità del dominio e i modelli di *dati*_G utilizzati nel *Core*_G.



- I $Port_G$ stabiliscono le interfacce attraverso cui avviene l'interazione tra il $Core_G$ e l'ambiente esterno. Essi stabiliscono dei contratti che specificano cosa deve essere fatto e vengono utilizzati dai $Core_G$ per interagire con l'ambiente esterno;
- Gli $Adapter_G$ implementano concretamente le interfacce definite dalle $Port_G$ e fungono da intermediari tra il $Core_G$ e le dipendenze esterne. Gli $Adapter_G$ si occupano della traduzione dei $dati_G$ e delle richieste tra il dominio e l'ambiente esterno;
- I $Repository_G$ implementa la logica di persistenza del $sistema_G$, occupandosi della memorizzazione e del recupero delle informazioni tramite l'interazione con $database_G$ o altri meccanismi di storage. I $dati_G$ trattati all'interno del $Repository_G$ sono rappresentati dalle $Entity_G$, le quali fungono da strato intermedio tra i modelli di business e la rappresentazione persistente delle informazioni.

L'utilizzo dei $Port_G$ e degli $Adapter_G$ garantiscono modularità, manutenibilità e flessibilità del $sistema_G$. Esse infatti isolano il $Core_G$ dall'ambiente esterno, favorendo testabilità e flessibilità, dando la possibilità di testare il $Core_G$ utilizzando $mock_G$ o $stub_G$ delle $Port_G$ e di cambiare o sostituire gli $Adapter_G$ senza modificare il $core_G$.

Il $backend_G$ è composto da due parti principali:

- $Scraper_G$: utilizza il $framework_G Scrapy_G$ per gestire il $processo_G$ di scraping ed è progettato per raccogliere, elaborare e salvare dati relativi a prodotti, file PDF e FAQ_G dal sito web aziendale;
- $Supabase_G$ functions: rappresentano il cuore della logica di business del $sistema_G$. Si occupano di gestire la logica di business implementando casi d'uso per operazioni come la gestione di documenti, prodotti, embedding, messaggi e $chunk_G$ e interagiscono con il mondo esterno utilizzano $Adapter_G$ e $Repository_G$ per comunicare con il $database_G$, modelli AI_G e altre fonti di $dati_G$.

3.6.2.1 Componenti principali dello Scraper

Core

• Models:

- $DbUploadOperationResponse$: rappresenta il risultato di un'operazione di caricamento di un file nel $database_G$. Essa espone i metodi:
 - * $__init__(self, success: bool, objID: str, message: str)$: costruttore della classe, utilizzato per inizializzare un'istanza di $DbUploadOperationResponse$;
 - * $get_success(self) \rightarrow bool$: restituisce il valore dell'attributo $success$, che indica se l'operazione di caricamento è stata eseguita con successo;
 - * $get_objID(self) \rightarrow str$: restituisce il valore dell'attributo $objID$, che rappresenta l'ID dell'oggetto caricato nel $database_G$;
 - * $get_message(self) \rightarrow str$: restituisce il valore dell'attributo $message$, che rappresenta un messaggio associato all'operazione di caricamento.
- $FilePdf$: rappresenta un file PDF con informazioni come il percorso, l'URL e un eventuale ID associato. Essa espone i metodi:
 - * $__init__(self, path: str, url: str, objID: str = "")$: costruttore della classe, utilizzato per inizializzare un'istanza di $FilePdf$;
 - * $get_path(self) \rightarrow str$: restituisce il valore dell'attributo $path$, che rappresenta il percorso del file PDF;
 - * $get_objID(self) \rightarrow str$: restituisce il valore dell'attributo $objID$, che rappresenta l'ID dell'oggetto associato al file PDF;
 - * $get_url(self) \rightarrow str$: restituisce il valore dell'attributo url , che rappresenta l'URL associato al file PDF;



- * `get_name(self) -> str`: restituisce il nome del file PDF;
- * `set_objID(self, objID: str)`: imposta un nuovo valore per l'attributo `objID`.
- `DbCheckOperationResponse`: rappresenta il risultato di un'operazione di verifica, come controllare se un file è aggiornato nel `databaseG`. Essa espone i metodi:
 - * `__init__(self, success: bool, message: bool)`: costruttore della classe, utilizzato per inizializzare un'istanza di `DbCheckOperationResponse`;
 - * `get_success(self) -> bool`: restituisce il valore dell'attributo privato `__success`, che indica se l'operazione è stata eseguita con successo;
 - * `get_message(self) -> bool`: restituisce il valore dell'attributo privato `__message`, che rappresenta un messaggio associato all'operazione;
 - * `__eq__(self, other) -> bool`: definisce il comportamento dell'operatore di uguaglianza per gli oggetti della classe.
- `DbInsertOperationResponse`: rappresenta il risultato di un'operazione di inserimento nel `databaseG`. Essa espone i metodi:
 - * `__init__(self, success: bool, message: bool)`: costruttore della classe, utilizzato per inizializzare un'istanza di `DbInsertOperationResponse`;
 - * `get_success(self) -> bool`: restituisce il valore dell'attributo privato `__success`, che indica se l'operazione di inserimento è stata eseguita con successo;
 - * `get_message(self) -> bool`: restituisce il valore dell'attributo privato `__message`, che rappresenta un messaggio associato all'operazione di inserimento;
 - * `__eq__(self, other) -> bool`: definisce il comportamento dell'operatore di uguaglianza per gli oggetti della classe.
- `Faq`: rappresenta una `FAQG` (Frequently Asked Question) associata a un prodotto. Questa classe incapsula le informazioni relative a una `FAQG`, come l'ID del prodotto, la domanda e la risposta. Essa espone i metodi:
 - * `__init__(self, productID: str, question: str, answer: str)`: costruttore della classe, utilizzato per inizializzare un'istanza di `Faq`;
 - * `get_productID(self) -> str`: restituisce il valore dell'attributo `productID`, che rappresenta l'ID del prodotto associato alla `FAQG`;
 - * `get_question(self) -> str`: restituisce il valore dell'attributo `question`, che rappresenta la domanda della `FAQG`;
 - * `get_answer(self) -> str`: restituisce il valore dell'attributo `answer`, che rappresenta la risposta alla domanda della `FAQG`.
- `Product`: rappresenta un prodotto nel dominio dell'`applicazioneG`. Questa classe incapsula le informazioni principali relative a un prodotto, come l'ID, il nome, la descrizione e i dati ETIM. Essa espone i metodi:
 - * `__init__(self, id: str, name: str, description: str, etim: str)`: costruttore della classe, utilizzato per inizializzare un'istanza di `Product`;
 - * `get_id(self) -> str`: restituisce il valore dell'attributo privato `__id`, che rappresenta l'ID univoco del prodotto;
 - * `get_name(self) -> str`: restituisce il valore dell'attributo privato `__name`, che rappresenta il nome del prodotto;
 - * `get_description(self) -> str`: restituisce il valore dell'attributo privato `__description`, che rappresenta la descrizione del prodotto;
 - * `get_etim(self) -> str`: restituisce il valore dell'attributo privato `__etim`, che rappresenta i dati ETIM del prodotto.

• **UseCase:**



- *CheckUpdatedFileUseCase*: definisce un'interfaccia astratta per il caso d'uso di verifica se un file è già aggiornato nel $database_G$. Essa definisce il metodo astratto *check_updated_file(self, file: FilePdf) -> DbCheckOperationResponse* il quale è progettato per verificare se un file è già aggiornato nel $sistema_G$;
- *EndUpdateUseCase*: definisce un'interfaccia astratta per il caso d'uso di terminare un $processo_G$ di aggiornamento. Essa definisce il metodo astratto *end_update(self) -> DbInsertOperationResponse*, progettato per concludere un $sistema_G$ di aggiornamento;
- *InsertAssociationProductFileUseCase*: definisce un'interfaccia astratta per il caso d'uso di inserire un'associazione tra un prodotto e un file nel $database_G$. Essa definisce il metodo *insert_association_product_file(self, product: Product, file: FilePdf) -> DbInsertOperationResponse*, progettato per inserire un'associazione tra un prodotto e un file nel $database_G$;
- *InsertFaqUseCase*: definisce un'interfaccia astratta per il caso d'uso di inserire una FAQ nel $database_G$. Essa definisce il metodo astratto *sistema_G*, progettato per inserire una $sistema_G$ nel $database_G$;
- *InsertFileUseCase*: definisce un'interfaccia astratta per il caso d'uso di inserire un file nel $database_G$. Essa definisce il metodo astratto *insert_file(self, file: FilePdf) -> DbInsertOperationResponse*, progettato per inserire un file nel $database_G$;
- *InsertProductUseCase*: definisce un'interfaccia astratta per il caso d'uso di inserire un prodotto nel $database_G$. Essa definisce il metodo astratto *insert_product(self, product: Product) -> DbInsertOperationResponse*, progettato per inserire un prodotto nel database.
- *UploadFileUseCase*: definisce un'interfaccia astratta per il caso d'uso di caricare un file nel $database_G$ o in un $sistema_G$ di archiviazione. Essa definisce il metodo astratto *UploadFileUseCase(ABC)*, progettato per caricare un file nel $sistema_G$.

• **Services:**

- *CheckUpdatedFileService*: servizio che implementa il caso d'uso definito nell'interfaccia *CheckUpdatedFileUseCase*. Il suo scopo è verificare se un file è aggiornato nel $database_G$, delegando l'operazione alla porta *CheckUpdatedFilePort*. Questa classe implementa i metodi:
 - * *__init__(self, check_updated_file_port: CheckUpdatedFilePort)*: costruttore della classe, utilizzato per inizializzare un'istanza di *CheckUpdatedFileService*. Inoltre esso configura la porta *CheckUpdatedFilePort*;
 - * *check_updated_file(self, file: FilePdf) -> DbCheckOperationResponse*: implementa il metodo astratto definito in *CheckUpdatedFileUseCase*. Esso verifica se un file è aggiornato nel $database_G$, delegando l'operazione alla porta *CheckUpdatedFilePort*.
- *EndUpdateService*: servizio che implementa il caso d'uso definito nell'interfaccia *EndUpdateUseCase*. Il suo scopo è gestire la logica per terminare un processo di aggiornamento, delegando l'operazione alla porta *EndUpdatePort*. Questa classe implementa i metodi:
 - * *__init__(self, end_update_port: EndUpdatePort)*: costruttore della classe e configura la porta *EndUpdatePort*;
 - * *end_update(self) -> DbInsertOperationResponse*: implementa il metodo astratto definito in *EndUpdateUseCase*. Conclude il $processo_G$ di aggiornamento, delegando l'operazione alla porta *EndUpdatePort*.
- *InsertAssociationProductFileService*: servizio che implementa il caso d'uso definito nell'interfaccia *InsertAssociationProductFileUseCase*. Il suo scopo è gestire la logica per inserire un'associazione tra un prodotto e un file nel $database_G$, delegando l'operazione alla porta *InsertAssociationProductFilePort*. Questa classe implementa i metodi:
 - * *__init__(self, insert_association_product_file_port: InsertAssociationProductFilePort)*: costruttore della classe e configura la porta *InsertAssociationProductFilePort*;



- * *insert_association_product_file(self, product: Product, file: FilePdf) -> DbInsertOperationResponse*: implementa il metodo astratto definito in *InsertAssociationProductFileUseCase*. Inserisce un'associazione tra un prodotto e un file nel *database_G*, delegando l'operazione alla porta *InsertAssociationProductFilePort*.
- *InsertFaqService*: servizio che implementa il caso d'uso definito nell'interfaccia *InsertFaqUseCase*. Il suo scopo è gestire la logica per inserire una *FAQ_G* (Frequently Asked Question) nel *database_G*, delegando l'operazione alla porta *InsertFaqPort*. Questa classe implementa i metodi:
 - * *__init__(self, insert_faq_port: InsertFaqPort)*: costruttore della classe e configura la porta *InsertFaqPort*;
 - * *insert_faq(self, faq: Faq) -> DbInsertOperationResponse*: implementa il metodo astratto definito in *InsertFaqUseCase*. Inserisce una *FAQ_G* nel *database_G*, delegando l'operazione alla porta *InsertFaqPort*.
- *InsertFileService*: servizio che implementa il caso d'uso definito nell'interfaccia *InsertFileUseCase*. Il suo scopo è gestire la logica per inserire un file nel *database_G*, delegando l'operazione alla porta *InsertFilePort*. Questa classe implementa i metodi:
 - * *__init__(self, insert_file_port: InsertFilePort)*: costruttore della classe e configura la porta *InsertFilePort*;
 - * *insert_file(self, file: FilePdf) -> DbInsertOperationResponse*: implementa il metodo astratto definito in *InsertFileUseCase*. Inserisce un file all'interno del *database_G*, delegando l'operazione alla porta *InsertFilePort*.
- *Insert ProductService*: servizio che implementa il caso d'uso definito nell'interfaccia *InsertProductUseCase*. Il suo scopo è gestire la logica per inserire un prodotto nel *database_G*, delegando l'operazione alla porta *InsertProductPort*. Questa classe implementa i metodi:
 - * *__init__(self, insert_product_port: InsertProductPort)*: costruttore della classe e configura la porta *InsertProductPort*;
 - * *insert_product(self, product: Product) -> DbInsertOperationResponse*: implementa il metodo astratto definito in *InsertProductUseCase*. Inserisce un prodotto all'interno del *database_G*, delegando l'operazione alla porta *InsertFilePort*.
- *UploadFileService*: servizio che implementa il caso d'uso definito nell'interfaccia *UploadFileUseCase*. Il suo scopo è gestire la logica per caricare un file nel *database_G*, delegando l'operazione alla porta *UploadFilePort*. Questa classe implementa i metodi:
 - * *__init__(self, upload_file_port: UploadFilePort)*: costruttore della classe e configura la porta *UploadFilePort*;
 - * *upload_file(self, file: FilePdf) -> DbUploadOperationResponse*: implementa il metodo astratto definito in *uploadFileUseCase*. Carica un file all'interno del *database_G*, delegando l'operazione alla porta *UploadFilePort*.

Port

- *CheckUpdatedFilePort*: interfaccia astratta che definisce il contratto attraverso il quale il *CheckUpdatedFileService* interagisce con il mondo esterno, per verificare se un file è aggiornato senza modificare la logica di business. Essa definisce il metodo astratto *check_updated_file(self, file: FilePdf) -> DbCheckOperationResponse* il quale è progettato per verificare se un file è già aggiornato nel *sistema_G*;
- *EndUpdatePort*: interfaccia astratta che definisce il contratto attraverso il quale il *EndUpdateService* interagisce con il mondo esterno, per terminare il *processo_G* di aggiornamento senza modificare la logica di business. Essa definisce il metodo astratto *end_update(self) -> DbInsertOperationResponse* che determina se il *processo_G* di aggiornamento è terminato o meno;



- *InsertAssociationProductFilePort*: interfaccia astratta che definisce il contratto attraverso il quale il *InsertAssociationProductFileService* interagisce con il mondo esterno, per inserire un'associazione tra un prodotto e un file nel *database_G* senza modificare la logica di business. Essa definisce il metodo astratto *insert_association_product_file(self, product: Product, file: FilePdf) -> DblInsertOperationResponse*;
- *InsertFaqPort*: interfaccia astratta che definisce il contratto attraverso il quale il *InsertFaqService* interagisce con il mondo esterno, per inserire una *FAQ_G* (Frequently Asked Question) nel *database_G*; Essa definisce il metodo *insert_faq(self, faq: Faq) -> DblInsertOperationResponse* che determina se la *FAQ_G* è stata inserita o meno;
- *InsertFilePort*: interfaccia astratta che definisce il contratto attraverso il quale il *InsertFileService* interagisce con il mondo esterno, per inserire un file nel *database_G*; Essa definisce il metodo *insert_file(self, file: FilePdf) -> DblInsertOperationResponse* che determina se il file è stato effettivamente inserito nel *database_G*;
- *InsertProductPort*: interfaccia astratta che definisce il contratto attraverso il quale il *InsertProductService* interagisce con il mondo esterno, per inserire un prodotto nel *database_G*; Essa definisce il metodo *insert_product(self, product: Product) -> DblInsertOperationResponse* che determina se il prodotto è stato inserito o meno nel *database_G*;
- *UploadFilePort*: interfaccia astratta che definisce il contratto attraverso il quale il *UploadFileService* interagisce con il mondo esterno, per caricare un file in un *database_G*. Essa definisce il metodo *upload_file(self, file: FilePdf) -> DbUploadOperationResponse* che determina se il file è stato caricato o meno nel *database_G*.

Adapter

- *SupabaseAdapter*: adattatore che implementa diverse porte (interfacce) per interagire con un'istanza di *Supabase_G*, un *database_G* o sistema esterno. Questa classe funge da ponte tra il *Core_G* dell'*applicazione_G* e *Supabase_G*, traducendo i *dati_G* e le operazioni del dominio in un formato comprensibile per *Supabase_G* e viceversa.

I metodi implementati dall'*adapter_G* sono:

- *__init__(self, repository: SupabaseRepository)*: costruttore della classe;
- *insert_product(self, product: Product) -> DblInsertOperationResponse*: si occupa di inserire un prodotto nel *database_G* utilizzando il *repository_G*;
- *check_updated_file(self, file: FilePdf) -> DbCheckOperationResponse*: si occupa di verificare se un file è aggiornato nel *database_G* utilizzando il *repository_G*;
- *upload_file(self, file: FilePdf) -> DbUploadOperationResponse*: si occupa di caricare un file nel *database_G* utilizzando il *repository_G*;
- *insert_file(self, file: FilePdf) -> DblInsertOperationResponse*: si occupa di inserire un file nel *database_G* utilizzando il *repository_G*;
- *insert_association_product_file(self, product: Product, file: FilePdf) -> DblInsertOperationResponse*: si occupa di inserire un'associazione tra un prodotto e un file nel *database_G* utilizzando il *repository_G*;
- *insert_faq(self, faq: Faq) -> DblInsertOperationResponse*: si occupa di inserire una *FAQ_G* nel *database_G* utilizzando il *repository_G*;
- *end_update(self) -> DblInsertOperationResponse*: si occupa di terminare il *processo_G* di aggiornamento nel *database_G* utilizzando il *repository_G*;



- `__supabase_product_converter(self, product: Product) -> SupabaseProduct`: si occupa di convertire un oggetto `Product` in un oggetto `SupabaseProduct`, che rappresenta il prodotto nel formato richiesto da `SupabaseG`;
- `__supabase_file_converter(self, file: FilePdf) -> SupabaseFile`: si occupa di convertire un oggetto `FilePdf` in un oggetto `SupabaseFile`, che rappresenta il file nel formato richiesto da `SupabaseG`;
- `__db_insert_operation_response_converter(self, supabase_response: SupabaseInsertOperationResponse) -> DbInsertOperationResponse`: si occupa di convertire un oggetto `SupabaseInsertOperationResponse` in un oggetto `DbInsertOperationResponse`, che rappresenta la risposta dell'operazione di inserimento nel `databaseG`;
- `__db_check_operation_response_converter(self, supabase_response: SupabaseCheckOperationResponse) -> DbCheckOperationResponse`: si occupa di convertire un oggetto `SupabaseCheckOperationResponse` in un oggetto `DbCheckOperationResponse`, che rappresenta la risposta dell'operazione di verifica nel `databaseG`;
- `__db_upload_operation_response_converter(self, supabase_response: SupabaseUploadOperationResponse) -> DbUploadOperationResponse`: si occupa di convertire un oggetto `SupabaseUploadOperationResponse` in un oggetto `DbUploadOperationResponse`, che rappresenta la risposta dell'operazione di caricamento nel `databaseG`;
- `__supabase_faq_converter(self, faq: Faq) -> SupabaseFaq`: si occupa di convertire un oggetto `Faq` in un oggetto `SupabaseFaq`, che rappresenta la `FAQG` nel formato richiesto da `SupabaseG`.

Repository

- `SupabaseRepository`: è un `RepositoryG` che fornisce metodi per interagire direttamente con il `databaseG`. Questa classe funge da livello di accesso ai `datiG`, eseguendo operazioni `CRUDG` (Create, Read, Update, Delete) su tabelle specifiche del `databaseG`. È utilizzata dagli adattatori per implementare le porte definite nel `CoreG` dell'applicazione_G.

I metodi implementati dal `RepositoryG` sono:

- `__init__(self)`: costruttore della classe;
- `insert_product(self, product: SupabaseProduct) -> SupabaseInsertOperationResponse`: si occupa di inserire un prodotto nel `databaseG` e restituisce la risposta dell'operazione;
- `check_updated_file(self, file: SupabaseFile) -> SupabaseCheckOperationResponse`: si occupa di verificare se un file è aggiornato nel `databaseG` e restituisce la risposta dell'operazione;
- `upload_file(self, file: SupabaseFile) -> SupabaseUploadOperationResponse`: si occupa di caricare un file nel `databaseG` e restituisce la risposta dell'operazione;
- `insert_file(self, file: SupabaseFile) -> SupabaseInsertOperationResponse`: si occupa di inserire un file nel `databaseG` e restituisce la risposta dell'operazione;
- `insert_association_product_file(self, product: SupabaseProduct, file: SupabaseFile) -> SupabaseInsertOperationResponse`: si occupa di inserire un'associazione tra un prodotto e un file nel `databaseG` e restituisce la risposta dell'operazione;
- `insert_faq(self, faq: SupabaseFaq) -> SupabaseInsertOperationResponse`: si occupa di inserire una `FAQG` nel `databaseG` e restituisce la risposta dell'operazione;
- `end_update(self) -> SupabaseInsertOperationResponse`: si occupa di terminare il `processoG` di aggiornamento nel `databaseG` e restituisce la risposta dell'operazione.

3.6.2.2 Componenti principali delle Supabase functions

Core

- **Models:**



- *AIAnswer*: rappresenta una risposta generata da un *sistema_G* di *AI_G*. Questa classe incapsula i dettagli della risposta, come il successo dell'operazione e il contenuto della risposta stessa. Essa espone i metodi:
 - * *getSuccess(): boolean*: restituisce il valore dell'attributo privato *success*, che indica se l'operazione di generazione della risposta è stata eseguita con successo;
 - * *getAnswer(): string*: restituisce il valore dell'attributo privato *answer*, che rappresenta la risposta generata dal *sistema_G* di *AI_G*.
- *AIEmbedding*: rappresenta il risultato di un'operazione di embedding generata da un *sistema_G* di *AI_G*. Essa espone i metodi:
 - * *getSuccess(): boolean*: restituisce il valore dell'attributo privato *success*, che indica se l'operazione di generazione dell'embedding è stata eseguita con successo;
 - * *getAnswer(): string*: restituisce il valore dell'attributo privato *answer*, che rappresenta l'*sistema_G* generato dal modello di embedding.
- *AIPrompt*: rappresenta un *prompt_G* utilizzato per interagire con un modello di intelligenza artificiale. Essa esponde i metodi:
 - * *getPrompt(): OpenAI.Chat.Completions.ChatCompletionMessageParam[]*: restituisce il valore dell'attributo privato *prompt*, che rappresenta il *prompt_G* da inviare al modello di *AI_G*;
- *AIQuestion*: rappresenta una domanda da inviare a un *sistema_G* di *AI_G*. Essa espone i metodi:
 - * *getQuestion(): string*: restituisce il valore dell'attributo privato *question*, che rappresenta la domanda da inviare al *sistema_G* di *AI_G*;
- *Chat e Message*: rappresentano rispettivamente una conversazione e un messaggio all'interno di una chat. Sono utilizzate per gestire e organizzare i *dati_G* relativi alle interazioni, come domande, risposte, prodotti associati e altre informazioni.

Chat espone i metodi:

 - * *getID(): string*: restituisce il valore dell'attributo privato *id*, che rappresenta l'ID univoco della chat;
 - * *getMessages(): Message[]*: restituisce il valore dell'attributo privato *messages*, che rappresenta la lista dei messaggi all'interno della chat;
 - * *getLastMessage(): Message*: restituisce il valore *lastMessage*, che rappresenta l'ultimo messaggio all'interno della chat.

Message espone i metodi:

 - * *getID(): string*: restituisce il valore dell'attributo privato *id*, che rappresenta l'ID univoco del messaggio;
 - * *getChatID(): string*: restituisce il valore dell'attributo privato *chatID*, che rappresenta l'ID della chat a cui appartiene il messaggio;
 - * *getQuestion(): string*: restituisce il valore dell'attributo privato *question*, che rappresenta la domanda inviata nel messaggio;
 - * *getAnswer(): string*: restituisce il valore dell'attributo privato *answer*, che rappresenta la risposta ricevuta nel messaggio;
 - * *getDate(): Date*: restituisce il valore dell'attributo privato *date*, che rappresenta la data e l'ora in cui è stato inviato il messaggio;
 - * *getProductNames(): string[]*: restituisce il valore dell'attributo privato *productNames*, che rappresenta i nomi dei prodotti associati al messaggio;
 - * *getProductIDs(): string[]*: restituisce il valore dell'attributo privato *productIDs*, che rappresenta gli ID dei prodotti associati al messaggio;
 - * *getEmbedding(): number[]*: restituisce il valore dell'attributo privato *embedding*, che rappresenta l'*embedding_G* associato al messaggio;



- * `getLanguage(): string`: restituisce il valore dell'attributo privato `language`, che rappresenta la lingua del messaggio;
 - * `setQuestion(question: string): void`: imposta il valore dell'attributo privato `question` con la domanda fornita;
 - * `setAnswer(answer: string): void`: imposta il valore dell'attributo privato `answer` con la risposta fornita;
 - * `setProductNames(productNames: string[]): void`: imposta il valore dell'attributo privato `productNames` con i nomi dei prodotti forniti;
 - * `setProductIDs(productIDs: string[]): void`: imposta il valore dell'attributo privato `productIDs` con gli ID dei prodotti forniti;
 - * `setEmbedding(embedding: number[]): void`: imposta il valore dell'attributo privato `embedding` con l'`embeddingG` fornito.
- `DBInsertResponse`: rappresenta il risultato di un'operazione di inserimento nel `databaseG`. Essa espone i metodi:
- * `getSuccess(): boolean`: restituisce il valore dell'attributo privato `success`, che indica se l'operazione di inserimento è stata eseguita con successo;
 - * `getAnswer(): string`: restituisce il valore dell'attributo privato `answer`.
- `DbUpsertResponse`: rappresenta il risultato di un'operazione di upsert (update o insert) nel `databaseG`. Essa espone i metodi:
- * `public getSuccess(): boolean`: restituisce il valore dell'attributo privato `success`, che indica se l'operazione di upsert è stata eseguita con successo;
 - * `getMessage(): boolean`: restituisce il valore dell'attributo privato `message`, che rappresenta un messaggio associato all'operazione di upsert.
- `Document` e `Chunk`: rappresentano rispettivamente un documento e i suoi `chunkG`. Sono utilizzate per gestire i `datiG` relativi ai documenti, come metadati, contenuto, URL, e per suddividere i documenti in parti più piccole (`chunkG`) con informazioni aggiuntive come `embeddingG`. `Document` espone i metodi:
- * `getName(): string`: restituisce il valore dell'attributo privato `name`, che rappresenta il nome del documento;
 - * `getId(): string`: restituisce il valore dell'attributo privato `id`, che rappresenta l'ID univoco del documento;
 - * `getUpdatedAt(): string`: restituisce il valore dell'attributo privato `updatedAt`, che rappresenta la data e l'ora dell'ultimo aggiornamento del documento;
 - * `getCreatedAt(): string`: restituisce il valore dell'attributo privato `createdAt`, che rappresenta la data e l'ora di creazione del documento;
 - * `getLastAccessedAt(): string`: restituisce il valore dell'attributo privato `lastAccessedAt`, che rappresenta la data e l'ora dell'ultimo accesso al documento;
 - * `getMetadata(): Record<string, string | number>`: restituisce il valore dell'attributo privato `metadata`, che rappresenta i metadati associati al documento;
 - * `getBlobData(): Blob | undefined`: restituisce il valore dell'attributo privato `blobData`, che rappresenta i dati binari del documento;
 - * `getUrl(): string`: restituisce il valore dell'attributo privato `url`, che rappresenta l'URL del documento;
 - * `getNChunks(): number`: restituisce il valore dell'attributo privato `nChunks`, che rappresenta il numero di `chunkG` associati al documento;
 - * `setName(name: string): void`: imposta il valore dell'attributo privato `name` con il nome fornito;
 - * `setId(id: string): void`: imposta il valore dell'attributo privato `id` con l'ID fornito;



- * `setUpdatedAt(updated_at: string): void`: imposta il valore dell'attributo privato `updatedAt` con la data e l'ora fornite;
- * `setCreatedAt(created_at: string): void`: imposta il valore dell'attributo privato `createdAt` con la data e l'ora fornite;
- * `setLastAccessedAt(last_accessed_at: string): void`: imposta il valore dell'attributo privato `lastAccessedAt` con la data e l'ora fornite;
- * `setMetadata(metadata: Record<string, string | number>): void`: imposta il valore dell'attributo privato `metadata` con i metadati forniti;
- * `setBlobData(blobData: Blob): void`: imposta il valore dell'attributo privato `blobData` con i dati binari forniti;
- * `setUrl(url: string): void`: imposta il valore dell'attributo privato `url` con l'URL fornito;
- * `setNChunks(nChunks: number): void`: imposta il valore dell'attributo privato `nChunks` con il numero di $chunk_G$ fornito.

Chunk espone i metodi:

- * `getContent(): string`: restituisce il valore dell'attributo privato `content`, che rappresenta il contenuto del $chunk_G$;
 - * `getDocument(): string`: restituisce il valore dell'attributo privato `document`, che rappresenta il documento associato al $chunk_G$;
 - * `getNumber(): number`: restituisce il valore dell'attributo privato `number`, che rappresenta il numero del $chunk_G$;
 - * `getEmbedding(): number[]`: restituisce il valore dell'attributo privato `embedding`, che rappresenta l' $embedding_G$ associato al $chunk_G$;
 - * `setContent(content: string): void`: imposta il valore dell'attributo privato `content` con il contenuto fornito;
 - * `setDocument(document: string): void`: imposta il valore dell'attributo privato `document` con il documento fornito;
 - * `setNumber(number: number): void`: imposta il valore dell'attributo privato `number` con il numero fornito;
 - * `setEmbedding(embedding: number[]): void`: imposta il valore dell'attributo privato `embedding` con l' $embedding_G$ fornito.
- *GeneralProductInfo*: rappresenta informazioni generali su un insieme di prodotti. Questa classe incapsula i nomi e gli ID dei prodotti e fornisce metodi per accedere a queste informazioni. Essa espone i metodi:
 - * `getNames(): string[]`: restituisce il valore dell'attributo privato `names`, che rappresenta i nomi dei prodotti;
 - * `getIds(): string[]`: restituisce il valore dell'attributo privato `ids`, che rappresenta gli ID dei prodotti.
 - *Product*: rappresenta un prodotto. Questa classe incapsula le informazioni principali relative a un prodotto, come l'ID, il nome, la descrizione e i dati ETIM, e fornisce metodi per accedere e modificare questi $dati_G$. Essa espone i metodi:
 - * `getID(): string`: restituisce il valore dell'attributo privato `id`, che rappresenta l'ID univoco del prodotto;
 - * `getName(): string`: restituisce il valore dell'attributo privato `name`, che rappresenta il nome del prodotto;
 - * `getDescription(): string`: restituisce il valore dell'attributo privato `description`, che rappresenta la descrizione del prodotto;
 - * `getEtim(): string`: restituisce il valore dell'attributo privato `etim`, che rappresenta i dati ETIM del prodotto;



- * `setName(name: string): void`: imposta il valore dell'attributo privato `name` con il nome fornito;
- * `setDescription(description: string): void`: imposta il valore dell'attributo privato `description` con la descrizione fornita.

• **Services:**

- `DownloadDocumentService`: implementa la logica per scaricare un documento. Essa implementa il metodo `downloadDocument(document: Document): Promise<Blob>` che restituisce un `blobG` del documento scaricato;
- `GenerateAnswerService`: implementa la logica per generare una risposta utilizzando un `sistemaG` di `AIG`. Essa implementa il metodo `generateAnswer(prompt: AIPrompt): Promise<AIAnswer>` che restituisce una risposta generata dal `sistemaG` di `AIG`;
- `GenerateEmbeddingService`: implementa la logica per generare un `embeddingG` di una domanda utilizzando un modello di `embeddingG`. Essa implementa il metodo `generateEmbedding(question: AIQuestion): Promise<AIEmbedding>` che restituisce un `embeddingG` generato dal modello di `embeddingG`;
- `GetAllProductService`: implementa la logica per ottenere informazioni su tutti i prodotti. Essa implementa il metodo `getAllProduct(): Promise<GeneralProductInfo>` che restituisce un oggetto `GeneralProductInfo` contenente i nomi e gli ID dei prodotti;
- `GetDocumentsService`: implementa la logica per ottenere i documenti associati a un prodotto. Essa implementa il metodo `getDocuments(product: Product): Promise<Document[]>` che restituisce un array di oggetti `Document` associati al prodotto fornito;
- `GetHistoryService`: implementa la logica per recuperare la cronologia di una chat. Essa implementa il metodo `getHistory(chat: Chat): Promise<Chat>` che restituisce un oggetto `Chat` contenente la cronologia dei messaggi associati alla chat fornita;
- `GetProductService`: implementa la logica per ottenere i dettagli di un prodotto specifico. Essa implementa il metodo `getProduct(productId: string|null, productName: string|null): Promise<Product>` che restituisce un oggetto `Product` contenente i dettagli del prodotto specificato;
- `ObtainAllMessagesService`: implementa la logica per ottenere tutti i messaggi disponibili. Essa implementa il metodo `obtainAllMessages(): Promise<Message[]>` che restituisce un array di oggetti `Message` contenenti tutti i messaggi disponibili;
- `ObtainDocumentService`: implementa la logica per ottenere i dettagli di un documento specifico. Essa implementa il metodo `obtainDocument(document: Document): Promise<Document>` che restituisce un oggetto `Document` contenente i dettagli del documento specificato;
- `ObtainSimilarChunkService`: implementa la logica per ottenere i `chunkG` più simili a un messaggio specifico, basandosi su una lista di documenti. Essa implementa il metodo `obtainSimilarChunk(message: Message, documents: Document[], nChunk: number): Promise<Chunk[]>` che restituisce un array di oggetti `Chunk` contenenti i `chunkG` più simili al messaggio fornito;
- `RemoveExtraChunkService`: implementa la logica per rimuovere i `chunkG` extra associati a un documento. Essa implementa il metodo `removeExtraChunk(document: Document): Promise<DBInsertResponse>` che restituisce un oggetto `DBInsertResponse` contenente il risultato dell'operazione di rimozione;
- `UpdateMessageService`: implementa la logica per aggiornare un messaggio esistente. Essa definisce il metodo astratto `updateMessage(message: Message): Promise<DBInsertResponse>` che restituisce un oggetto `DBInsertResponse` contenente il risultato dell'operazione di aggiornamento;



- *UpsertChunkService*: implementa la logica per eseguire un'operazione di upsert (update o insert) di un $chunk_G$. Essa implementa il metodo *upsertChunk(chunk: Chunk)*: *Promise<DBInsertResponse>* che restituisce un oggetto *DBInsertResponse* contenente il risultato dell'operazione di upsert.

Port

- *DownloadDocumentPort*: specifica il contratto per scaricare un documento. Essa definisce il metodo astratto *downloadDocument(document: Document)*: *Promise<Blob>* che restituisce un $blob_G$ del documento scaricato;
- *GenerateAnswerPort*: specifica il contratto per generare una risposta utilizzando un *sistema_G* di AI_G . Essa definisce il metodo astratto *generateAnswer(prompt: AIPrompt)*: *Promise<AIAnswer>* che restituisce una risposta generata dal *sistema_G* di AI_G ;
- *GenerateEmbeddingPort*: specifica il contratto per generare un *embedding_G* di una domanda utilizzando un modello di *embedding_G*. Essa definisce il metodo astratto *generateEmbedding(question: AIQuestion)*: *Promise<AIEmbedding>* che restituisce un *embedding_G* generato dal modello di *embedding_G*;
- *GetAllProductPort*: specifica il contratto per ottenere informazioni generali su tutti i prodotti. Essa definisce il metodo astratto *getAllProduct()*: *Promise<GeneralProductInfo>* che restituisce un oggetto *GeneralProductInfo* contenente i nomi e gli ID dei prodotti;
- *GetDocumentsPort*: specifica il contratto per ottenere i documenti associati a un prodotto. Essa definisce il metodo astratto *getDocuments(product: Product)*: *Promise<Document[]>* che restituisce un array di oggetti *Document* associati al prodotto fornito;
- *GetHistoryPort*: specifica il contratto per ottenere la cronologia di una chat. Essa definisce il metodo astratto *getHistory(chat: Chat)*: *Promise<Chat>* che restituisce un oggetto *Chat* contenente la cronologia dei messaggi associati alla chat fornita;
- *GetProductPort*: specifica il contratto per ottenere i dettagli di un prodotto specifico. Essa definisce il metodo astratto *getProduct(productName: string|null, productId: string|null)*: *Promise<Product>* che restituisce un oggetto *Product* contenente i dettagli del prodotto specificato;
- *ObtainAllMessagesPort*: specifica il contratto per ottenere tutti i messaggi disponibili. Essa definisce il metodo astratto *obtainAllMessages()*: *Promise<Message[]>* che restituisce un array di oggetti *Message* contenenti tutti i messaggi disponibili;
- *ObtainDocumentPort*: specifica il contratto per ottenere i dettagli di un documento specifico. Essa definisce il metodo astratto *obtainDocument(document: Document)*: *Promise<Document>* che restituisce un oggetto *Document* contenente i dettagli del documento specificato;
- *ObtainSimilarChunkPort*: specifica il contratto per ottenere i $chunk_G$ più simili a un messaggio specifico, basandosi su una lista di documenti. Essa definisce il metodo astratto *obtainSimilarChunk(message: Message, documents: Document[], nChunk: number)*: *Promise<Chunk[]>* che restituisce un array di oggetti *Chunk* contenenti i $chunk_G$ più simili al messaggio fornito;
- *RemoveExtraChunkPort*: specifica il contratto per rimuovere i $chunk_G$ extra associati a un documento. Essa definisce il metodo astratto *removeExtraChunk(document: Document)*: *Promise<DBInsertResponse>* che restituisce un oggetto *DBInsertResponse* contenente il risultato dell'operazione di rimozione;
- *UpdateMessagePort*: specifica il contratto per aggiornare un messaggio esistente. Essa definisce il metodo astratto *updateMessage(message: Message)*: *Promise<DBInsertResponse>* che restituisce un oggetto *DBInsertResponse* contenente il risultato dell'operazione di aggiornamento;



- *UpsertChunkPort*: specifica il contratto per eseguire un'operazione di upsert (update o insert) di un $chunk_G$. Essa definisce il metodo astratto *upsertChunk(chunk: Chunk): Promise<DBInsertResponse>* che restituisce un oggetto *DBInsertResponse* contenente il risultato dell'operazione di upsert.

Adapter

- *AnswerAdapter*: implementa l'interfaccia *GenerateAnswerPort*. Il suo scopo è fungere da ponte tra il *Core_G* dell'applicazione_G e un *Repository_G* specifico. Essa implementa i metodi:
 - *generateAnswer(prompt: AIPrompt): Promise<AIAnswer>*: riceve un $prompt_G$ e si occupa di restituire una risposta generata dal *sistema_G* di *AI_G*;
 - *convertToOllamaPrompt(prompt: AIPrompt): OllamaPrompt*: riceve un $prompt_G$ e lo converte in un formato compatibile con il *sistema_G* di *AI_G* di *Ollama_G*;
 - *convertToAIAnswer(answer: OllamaAnswer): AIAnswer*: riceve una risposta generata dal *sistema_G* di *AI_G* di *Ollama_G* e la converte in un oggetto *AIAnswer* compatibile con il *Core_G* dell'applicazione_G.
- *ChunkAdapter*: implementa le porte *UpsertChunkPort*, *RemoveExtraChunkPort* e *ObtainSimilarChunkPort*. Il suo scopo è fungere da ponte tra il *Core_G* dell'applicazione_G e il *Repository_G* *ChunkRepository*, traducendo i *dati_G* del dominio in entità specifiche di *Supabase_G* e viceversa. Essa implementa i metodi:
 - *obtainSimilarChunk(message: Message, documents: Document[], nChunk: number): Promise<Chunk[]>*: si occupa di ritornare i $chunk_G$ più simili al messaggio fornito. Viene inoltre fornito il documento a cui fanno riferimento i $chunk_G$ ed il numero di $chunk_G$ massimi da restituire;
 - *upsertChunk(chunk: Chunk): Promise<DBInsertResponse>*: si occupa di gestire l'operazione di upsert dei $chunk_G$ nel *database_G*;
 - *removeExtraChunk(document: Document): Promise<DBInsertResponse>*: si occupa della rimozione dei $chunk_G$ extra associati a un documento;
 - *convertMessageToSupabaseMessage(message: Message): SupabaseMessage*: si occupa di convertire un oggetto di tipo *Message* in un oggetto compatibile con *Supabase_G* (*SupabaseMessage*);
 - *convertDocumentToSupabaseDocument(document: Document): SupabaseDocument*: si occupa di convertire un oggetto di tipo *Document* in un oggetto compatibile con *Supabase_G* (*SupabaseDocument*);
 - *convertChunkToSupabaseChunk(chunk: Chunk): SupabaseChunk*: si occupa di convertire un oggetto di tipo *Chunk* in un oggetto compatibile con *Supabase_G* (*SupabaseChunk*);
 - *convertSupabaseChunkToChunk(supabaseChunk: SupabaseChunk): Chunk*: si occupa di convertire un oggetto di tipo *SupabaseChunk* in un oggetto compatibile all'interno del *Core_G* dell'applicazione_G;
 - *convertSupabaseInsertResponseToDBInsertResponse(supabaseInsertResponse: SupabaseInsertResponse): DBInsertResponse*: si occupa di convertire un oggetto di tipo *SupabaseInsertResponse* in un oggetto compatibile all'interno del *Core_G* dell'applicazione_G, ovvero *DBInsertResponse* il quale rappresenta un'operazione di inserimento nel *database_G*.
- *DocumentAdapter*: implementa le porte: *GetDocumentsPort* per ottenere i documenti associati a un prodotto, *ObtainDocumentPort* per ottenere i dettagli di un documento specifico e *DownloadDocumentPort* per scaricare un documento.
Funge da ponte tra il *Core_G* dell'applicazione_G e il *Repository_G* *DocumentRepository*, traducendo i modelli del dominio in entità specifiche di *Supabase_G* e viceversa. Essa implementa i metodi:



- `getDocuments(product: Product): Promise<Document[]>`: si occupa di ottenere i documenti associati a un prodotto;
 - `obtainDocument(document: Document): Promise<Document>`: si occupa di ottenere i dettagli di un documento specifico;
 - `downloadDocument(document: Document): Promise<Blob>`: si occupa di scaricare un documento specifico;
 - `convertProductToPostgresProduct(product: Product): SupabaseProduct`: si occupa di convertire un oggetto di tipo `Product` in un oggetto compatibile con `SupabaseG` (`SupabaseProduct`);
 - `convertDocumentToSupabaseDocument(document: Document): SupabaseDocument`: si occupa di convertire un oggetto di tipo `Document` in un oggetto compatibile con `SupabaseG` (`SupabaseDocument`);
 - `convertSupabaseDocumentToDocument(supabaseDocument: SupabaseDocument): Document`: si occupa di convertire un oggetto di tipo `SupabaseDocument` in un oggetto compatibile all'interno del `CoreG` dell'applicazione_G.
- `EmbeddingAdapter`: implementa la porta `GenerateEmbeddingPort`. Il suo scopo è fungere da ponte tra il `CoreG` dell'applicazione_G e il `RepositoryG EmbeddingRepository`, traducendo i modelli del dominio in entità specifiche di `OllamaG` e viceversa. Essa implementa i metodi:
- `generateEmbedding(question: AIQuestion): Promise<AIEmbedding>`: si occupa di generare un `embeddingG` di una domanda utilizzando un modello di `embeddingG`;
 - `convertToOllamaQuestion(question: AIQuestion): OllamaQuestion`: riceve una domanda e la converte in un formato compatibile con il `sistemaG` di `AIG` di `OllamaG`;
 - `convertToAIEmbedding(embedding: OllamaEmbedding): AIEmbedding`: `AIEmbedding`: riceve un `embeddingG` generato dal `sistemaG` di `AIG` di `OllamaG` e lo converte in un oggetto `AIEmbedding` compatibile con il `CoreG` dell'applicazione_G.
- `MessageAdapter`: implementa le porte `GetHistoryPort` e `UpdateMessagePort`. Il suo scopo è fungere da ponte tra il `CoreG` dell'applicazione_G e il `RepositoryG MessageRepository`, traducendo i modelli del dominio (ad esempio, `Message` e `Chat`) in entità specifiche di `SupabaseG` (ad esempio, `SupabaseMessage` e `SupabaseChat`) e viceversa. Essa implementa i metodi:
- `getHistory(chat: Chat): Promise<Chat>`: si occupa di ottenere la cronologia dei messaggi associati a una chat;
 - `updateMessage(message: Message): Promise<DBInsertResponse>`: si occupa di aggiornare un messaggio esistente;
 - `obtainAllMessages(): Promise<Message[]>`: si occupa di ottenere tutti i messaggi disponibili;
 - `convertChatToSupabaseChat(chat: Chat): SupabaseChat`: si occupa di convertire un oggetto di tipo `Chat` in un oggetto compatibile con `SupabaseG` (`SupabaseChat`);
 - `convertSupabaseChatToChat(supabaseChat: SupabaseChat): Chat`: si occupa di convertire un oggetto di tipo `SupabaseChat` in un oggetto compatibile all'interno del `CoreG` dell'applicazione_G;
 - `convertMessageToSupabaseMessage(message: Message): SupabaseMessage`: si occupa di convertire un oggetto di tipo `Message` in un oggetto compatibile con `SupabaseG` (`SupabaseMessage`);
 - `convertSupabaseMessageToMessage(supabaseMessage: SupabaseMessage): Message`: si occupa di convertire un oggetto di tipo `SupabaseMessage` in un oggetto compatibile all'interno del `CoreG` dell'applicazione_G;
 - `convertSupabaseInsertResponseToDBInsertResponse(supabaseInsertResponse: SupabaseInsertResponse): DBInsertResponse`: si occupa di convertire un oggetto di tipo `SupabaseInsertResponse` in un oggetto compatibile all'interno del `CoreG` dell'applicazione_G, ovvero `DBInsertResponse` il quale rappresenta un'operazione di inserimento nel `databaseG`.



- *ProductAdapter*: implementa le porte *GetAllProductPort* e *GetProductPort*. Il suo scopo è fungere da ponte tra il *Core_G* dell'*applicazione_G* e il *Repository_G ProductRepository*, traducendo i modelli del dominio (ad esempio, *GeneralProductInfo* e *Product*) in entità specifiche di *Supabase_G* (ad esempio, *SupabaseGeneralProductInfo* e *SupabaseProduct*) e viceversa. Essa implementa i metodi:
 - *getAllProduct(): Promise<GeneralProductInfo>*: si occupa di ottenere informazioni generali su tutti i prodotti;
 - *getProduct (productName: string|null, productId: string|null): Promise<Product>*: si occupa di ottenere i dettagli di un prodotto specifico;
 - *convertSupabaseGeneralProductInfo(product:SupabaseGeneralProductInfo): GeneralProductInfo*: si occupa di convertire un oggetto di tipo *SupabaseGeneralProductInfo* in un oggetto compatibile all'interno del *Core_G* dell'*applicazione_G*;
 - *convertPostgresProductToProduct(supabaseProduct: SupabaseProduct): Product*: si occupa di convertire un oggetto di tipo *SupabaseProduct* in un oggetto compatibile all'interno del *Core_G* dell'*applicazione_G*.

Repository

- *AnswerRepository*: interagisce con un *LLM_G* fornito da *OpenAI_G* per generare risposte basate su un *prompt_G*. Si trova nella parte infrastrutturale dell'*architettura esagonale_G* e fornisce un'implementazione concreta per la generazione di risposte. Essa implementa il metodo *generateAnswer(prompt: OllamaPrompt): Promise<OllamaAnswer>* che attraverso l'utilizzo di un modello di *AI_G*, si occupa di generare una risposta.
- *ChunkRepository*: interagisce con il *database_G* per eseguire operazioni relative ai *chunk_G*. Esso fornisce un'implementazione concreta per operazioni come il recupero di *chunk_G* simili, l'inserimento/aggiornamento di *chunk_G* e la rimozione di *chunk_G* extra. Essa implementa i metodi:
 - *obtainSimilarChunk(message: SupabaseMessage, documents: SupabaseDocument[], nChunk: number): Promise<SupabaseChunk[]>*: attraverso l'utilizzo della funzione *rpc_G match_manuale*, si occupa di ritornare i *chunk_G* più simili al messaggio fornito all'interno di una lista di documenti;
 - *upsertChunk(chunk: SupabaseChunk): Promise<SupabaseInsertResponse>*: si occupa di gestire l'operazione di upsert dei *chunk_G* nel *database_G*;
 - *removeExtraChunk(document: SupabaseDocument): Promise<SupabaseInsertResponse>*: si occupa della rimozione dei *chunk_G* extra associati a un documento.
- *DocumentRepository*: interagisce con *Supabase_G* e il suo *sistema_G* di storage per eseguire operazioni relative ai documenti. Fornisce un'implementazione concreta per operazioni come il recupero, l'ottenimento e il download di documenti. Essa implementa i metodi:
 - *getDocuments(product: SupabaseProduct): Promise<SupabaseDocument[]>*: si occupa di ottenere i documenti associati a un prodotto;
 - *obtainDocument(document: SupabaseDocument): Promise<SupabaseDocument>*: si occupa di ottenere i dettagli di un documento specifico;
 - *downloadDocument (document: SupabaseDocument): Promise<Blob>*: si occupa di scaricare un documento specifico;
- *EmbeddingRepository*: interagisce con l'*LLM_G* fornito da *OpenAI_G* per generare *embedding_G* basati su una domanda. Fornisce un'implementazione concreta per la generazione di *embedding_G*. Essa implementa il metodo *generateEmbedding(question: OllamaQuestion): Promise<OllamaEmbedding>* che attraverso l'utilizzo del modello di embedding, si occupa di generare l'*embedding_G* della domanda.



- *MessageRepository*: interagisce con il $database_G$ per eseguire operazioni relative ai messaggi e alle chat. Fornisce un'implementazione concreta per operazioni come il recupero della cronologia della chat, l'ottenimento di tutti i messaggi e l'aggiornamento di un messaggio. Essa implementa i metodi:
 - *getHistory(chat: SupabaseChat): Promise<SupabaseChat>*: si occupa di ottenere la cronologia dei messaggi associati a una chat attraverso la funzione $rpc_G.gelastmessages$;
 - *obtainAllMessages(): Promise<SupabaseMessage[]>*: si occupa di ottenere tutti i messaggi disponibili all'interno del $database_G$;
 - *updateMessage(message: SupabaseMessage): Promise<SupabaseInsertResponse>*: si occupa di aggiornare un messaggio esistente.
- *ProductRepository*: interagisce con il $database_G$ per eseguire operazioni relative ai prodotti. Fornisce un'implementazione concreta per operazioni come il recupero di tutti i prodotti e il recupero di un prodotto specifico. Essa implementa i metodi:
 - *getAllProduct(): Promise<SupabaseGeneralProductInfo[]>*: si occupa di ottenere informazioni generali, ovvero nome e id di tutti i prodotti;
 - *getProduct(name: string|null, id: string|null): Promise<SupabaseProduct>*: si occupa di ottenere i dettagli di un prodotto specifico verificando la sua effettiva presenza nel $database_G$.

3.6.2.3 Design pattern backend

Di seguito vengono riportati tutti i pattern architetturali utilizzati suddivisi per tipologia.

Pattern creazionali

- **Singleton pattern**: questo pattern è stato adottato per garantire che le classi *AnswerAdapter*, *ChunkAdapter*, *DocumentAdapter*, *EmbeddingAdapter*, *MessageAdapter* e *ProductAdapter* e le relative classi repository abbiano una sola istanza condivisa in tutto il sistema. Questo approccio consente di ottimizzare l'uso delle risorse e di mantenere uno stato coerente tra le diverse parti del sistema.

Pattern strutturali

Pattern comportamentali

- **Template pattern**: questo pattern è stato adottato per differenziare le metodologie di generazione delle risposte in funzione della tipologia di domanda ricevuta. Sono state individuate tre categorie di domande: *tecniche*, *generali* e di *confronto*. In base alla categoria identificata, viene dinamicamente adattato il $prompt_G$ fornito al modello LLM_G , al fine di ottimizzare la pertinenza e l'accuratezza della risposta generata.



3.6.3 Database

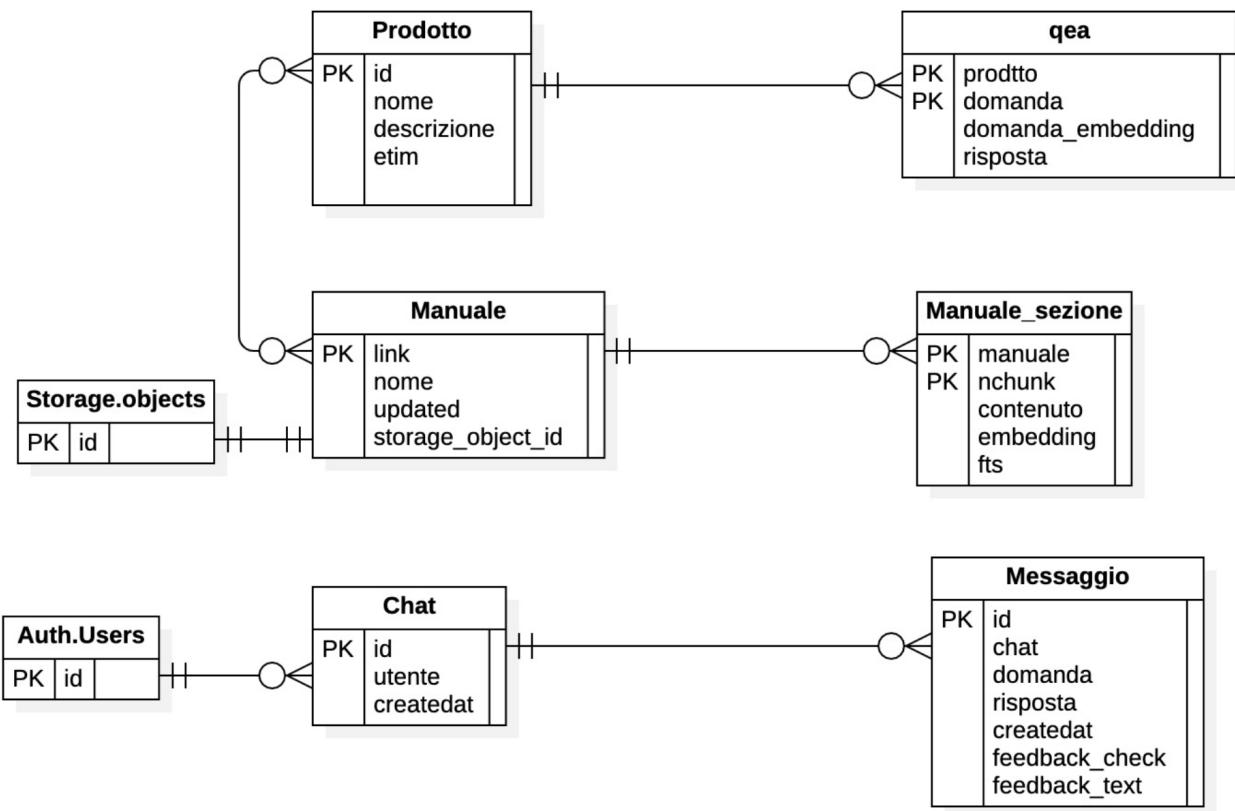


Figure 11: Diagramma ER Database

3.6.3.1 Entità principali

- **Prodotto**: contiene le informazioni principali relative ai prodotti disponibili sul sito web aziendale
 - *id*: codice alfanumerico univoco del prodotto;
 - *nome*: nome del prodotto;
 - *descrizione*: descrizione del prodotto;
 - *etim*: $dati_G$ ETIM del prodotto.
- **qea**: contiene un insieme di domande e risposte predefinite associate a ciascun prodotto
 - *prodotto*: riferimento al prodotto a cui la domanda/risposta è associata;
 - *domanda*: testo della domanda;
 - *domanda_embedding*: $embedding_G$ della domanda;
 - *risposta*: testo della risposta alla domanda.
- **Manuale**: contiene i riferimenti ai manuali tecnici associati ai prodotti presenti nel $sistema_G$
 - *link*: URL associato a ciascun manuale;
 - *nome*: nome del manuale;
 - *updated*: campo booleano che indica se il manuale è aggiornato;
 - *storage_object_id*: riferimento al file fisico del manuale salvato nello storage di $Supabase_G$.
- **Manuale_sezione**: contiene i $chunk_G$ relativi ai manuali



- *manuale*: manuale a cui appartiene il $chunk_G$;
 - *nchunk*: identificativo numerico univoco del $chunk_G$ relativo a un manuale;
 - *contenuto*: testo del $chunk_G$;
 - *embedding*: $embedding_G$ del testo del $chunk_G$;
 - *fts*: campo indicizzato per la full-text search.
- **Storage.objects**: contiene i file PDF relativi ai manuali
 - *id*: id univoco.
- 3.6.3.2 Autenticazione e chat**
- **Chat**: contiene le conversazioni tra un $utente_G$ e il $sistema_G$
 - *id*: id univoco della chat;
 - *utente*: riferimento all' $utente_G$ che ha avviato la sessione;
 - *createdat*: data e ora della creazione della chat.
 - **Messaggio**: contiene le singole interazioni avvenute all'interno di una sessione di chat tra l' $utente_G$ e il $sistema_G$
 - *id*: id univoco del messaggio;
 - *chat*: riferimento alla sessione di chat a cui il messaggio appartiene;
 - *domanda*: testo della domanda posta dall' $utente_G$;
 - *risposta*: testo della risposta generata dal $sistema_G$;
 - *createdat*: data e ora in cui il messaggio è stato registrato;
 - *feedback_check*: campo booleano che indica se è associato un $feedback_G$ sulla risposta;
 - *feedback_text*: eventuale testo del $feedback_G$ associato alla risposta.
 - **Auth.User**: contiene gli utenti autenticati all'interno del $sistema_G$
 - *id*: id univoco dell' $utente_G$.

4 Tracciamento dei requisiti

In questa sezione vengono elencati i requisiti funzionali, di vincolo e di qualità contenuti nelle tabelle presenti nella sezione "Requisiti funzionali", "Requisiti di vincolo" e "Requisiti di qualità" rispettivamente, del documento *Analisi dei Requisiti v2.0.0*. Nelle seguenti tabelle verrà aggiunta una colonna allo scopo di indicare se il requisito è stato soddisfatto o meno.

4.1 Tabella dei requisiti funzionali

Codice	Importanza	Descrizione	Check
RFO	Obbligatorio	Il $sistema_G$ deve poter lasciare gli $utenti_G$ fare richieste in lingua italiana	Soddisfatto
RF1	Opzionale	Il $sistema_G$ deve permettere agli $utenti_G$ di fare richieste in più lingue	Soddisfatto



RF2	Obbligatorio	Il <i>sistema_G</i> deve permettere agli <i>utenti_G</i> di digitare la domanda tramite tastiera	Soddisfatto
RF3	Obbligatorio	Il <i>sistema_G</i> deve permettere agli <i>utenti_G</i> di inviare le domande al <i>chatbot_G</i>	Soddisfatto
RF4	Opzionale	Il <i>sistema_G</i> deve permettere agli <i>utenti_G</i> di visualizzare data e ora di invio di ciascun messaggio	Soddisfatto
RF5	Obbligatorio	Il <i>sistema_G</i> deve prevedere un limite di caratteri per effettuare le richieste testuali da parte degli <i>utenti_G</i>	Soddisfatto
RF6	Obbligatorio	Il <i>sistema_G</i> deve rendere disponibile uno storico dei messaggi nella stessa conversazione	Soddisfatto
RF7	Opzionale	Il <i>sistema_G</i> deve verificare che l'input dell' <i>utente_G</i> non contenga argomenti che violino le aree proibite	Soddisfatto
RF8	Obbligatorio	Al termine di ogni sessione, l' <i>applicativo_G</i> deve dare la possibilità di salvataggio delle conversazioni che l' <i>utente_G</i> ha avuto in quella determinata sessione	Soddisfatto
RF9	Obbligatorio	Il <i>sistema_G</i> potrà salvare un limite massimo di conversazioni per ciascun <i>utente_G</i>	Soddisfatto
RF10	Obbligatorio	Il <i>sistema_G</i> deve dare la possibilità all' <i>utente_G</i> di cancellare conversazioni salvate	Soddisfatto
RF11	Obbligatorio	Il <i>sistema_G</i> deve dare la possibilità all' <i>utente_G</i> di visualizzare le conversazioni attive con il <i>chatbot_G</i>	Soddisfatto
RF12	Obbligatorio	Il <i>sistema_G</i> deve dare la possibilità all' <i>utente_G</i> di creare una nuova conversazione	Soddisfatto
RF13	Obbligatorio	L' <i>utente_G</i> deve ricevere una risposta predefinita quando pone una domanda relativa ad argomenti proibiti	Soddisfatto
RF14	Obbligatorio	L' <i>utente_G</i> deve ricevere una risposta predefinita quando pone una domanda relativa a contenuti non disponibili	Soddisfatto
RF15	Opzionale	Le risposte generate dall' <i>applicativo_G</i> devono mostrare dei link di riferimento alle fonti attinenti alla domanda posta dall' <i>utente_G</i>	Non soddisfatto



RF16	Opzionale	Il <i>sistema_G</i> deve essere in grado di suggerire le possibili future domande che l' <i>utente_G</i> potrebbe fare, a seguito della precedente interazione	Non soddisfatto
RF17	Opzionale	Il <i>sistema_G</i> deve avere un <i>sistema_G</i> di conversazione guidata per gli installatori al fine di aiutarli nella composizione di una domanda	Non soddisfatto
RF18	Obbligatorio	L' <i>applicativo_G</i> deve prevedere un <i>sistema_G</i> di feedback attraverso il quale ogni <i>utente_G</i> dopo ogni risposta può indicare se la risposta ottenuta è stata positiva o negativa	Soddisfatto
RF19	Obbligatorio	L' <i>applicativo_G</i> deve mostrare una sezione protetta da credenziali che contenga una <i>dashboard_G</i> per amministratori	Soddisfatto
RF20	Opzionale	L'amministratore deve essere in grado di visualizzare all'interno della <i>dashboard_G</i> il numero totale di richieste effettuate mediante conversazione libera o guidata	Soddisfatto
RF21	Opzionale	L'amministratore deve essere in grado di visualizzare all'interno della <i>dashboard_G</i> le statistiche relative ai termini più utilizzati nelle richieste	Soddisfatto
RF22	Obbligatorio	L'amministratore deve essere in grado di visualizzare all'interno della <i>dashboard_G</i> il grafico associato all'andamento mensile dell' <i>applicativo_G</i>	Soddisfatto
RF23	Obbligatorio	L'amministratore deve essere in grado di visualizzare all'interno della <i>dashboard_G</i> il numero di <i>feedback_G</i> positivi e negativi	Soddisfatto
RF24	Opzionale	L'amministratore deve essere in grado di visualizzare all'interno della <i>dashboard_G</i> il contenuto dei <i>feedback_G</i> positivi e negativi	Soddisfatto
RF25	Obbligatorio	L' <i>applicativo_G</i> deve essere in grado di navigare un elenco di prodotti all'interno del sito web dell'azienda fornitrice ed estrarre le informazioni utili	Soddisfatto
RF26	Obbligatorio	L' <i>applicativo_G</i> deve collezionare le informazioni utili in maniera strutturata all'interno di un <i>database_G</i>	Soddisfatto



RF27	Opzionale	Il $sistema_G$ deve poter scaricare file istruzioni in formato PDF	Soddisfatto
RF28	Obbligatorio	Il $sistema_G$ deve riuscire a ricavare informazioni utili dai PDF ed estrarre immagini degli schemi elettrici	Soddisfatto
RF29	Opzionale	Il $sistema_G$ deve prevedere la possibilità di aggiornamento nel $database_G$ di dati tecnici e pdf relativi a prodotti nuovi o aggiornati, facendo riferimento al sito vimar.com	Soddisfatto
RF30	Obbligatorio	L' $applicativo_G$ deve prevedere un $sistema_G$ di indicizzazione delle informazioni a partire dal $database_G$ in cui sono stati salvati i dati estratti dal sito web	Soddisfatto
RF31	Obbligatorio	L' LLM_G deve essere in grado di rispondere a richieste relative agli <i>impianti Smart_G</i>	Soddisfatto
RF32	Obbligatorio	L' LLM_G deve essere in grado di rispondere a richieste relative agli <i>impianti Domotici_G</i>	Soddisfatto
RF33	Opzionale	L' LLM_G deve essere in grado di rispondere a richieste relative agli <i>impianti Tradizionali_G</i>	Non soddisfatto
RF34	Obbligatorio	Il componente di interrogazione deve controllare che l'output dell' LLM_G non vada in conflitto con argomenti proibiti	Soddisfatto

Table 2: Tabella dei requisiti funzionali

4.2 Riepilogo requisiti funzionali

Il gruppo *Byte Your Dreams* ha soddisfatto in totale 31 requisiti funzionali su 35.

Tutti i requisiti funzionali obbligatori sono stati soddisfatti mentre i requisiti funzionali opzionali *RF15*, *RF16*, *RF17* e *RF33* non sono stati soddisfatti a causa di limiti temporali, preferendo concentrare le proprie risorse per il miglioramento o l'implementazione di altri requisiti ritenuti prioritari per il *progetto_G*.

