
# Code7Crusaders

Software Development Team

**Specifica Tecnica**

**Membri del Team:**
Enrico Cotti Cottini, Gabriele Di Pietro, Tommaso Diviesti
Francesco Lapenna, Matthew Pan, Eddy Pinarello, Filippo Rizzolo

| Ver | Data | Redattore | Verificatore | Descrizione |
|---|---|---|---|---|
| 1.0 | 01/04/2025 | Gabriele Di Pietro | Eddy Pinarello | Approvazione Documento |
| 0.7 | 31/03/2025 | Gabriele Di Pietro | Eddy Pinarello | Aggiornato stato dei Requisiti e aggiunti grafici |
| 0.6 | 26/03/2025 | Matthew Pan | Francesco Lapenna | Sezione Moduli |
| 0.5 | 26/03/2025 | Francesco Lapenna | Matthew Pan | Continuo Stesura sezione Architettura |
| 0.4 | 18/03/2025 | Matthew Pan | Francesco Lapenna | Stesura sezione Architettura |
| 0.3 | 12/03/2025 | Francesco Lapenna | Matthew Pan | Prima stesura Architettura |
| 0.2 | 5/03/2025 | Eddy Pinarello | Francesco Lapenna | Stesura sezioni 2 e 4 |
| 0.1 | 1/03/2025 | Eddy Pinarello | Francesco Lapenna | Prima stesura del documento |

# Indice



# Elenco delle tabelle

# Elenco delle figure

# 1 Introduzione

## 1.1 Scopo specifica tecnica

Questo documento è rivolto a tutti gli stakeholder coinvolti nel progetto Code7Crusaders, un chatbot B2B pensato per semplificare la ricerca di prodotti all'interno dei cataloghi dei distributori. Il documento fornisce una visione dettagliata dell'architettura del sistema, dei design pattern utilizzati, delle tecnologie adottate e delle scelte progettuali effettuate. Inoltre, include diagrammi UML delle classi e delle attività per descrivere il funzionamento del sistema in modo chiaro e strutturato.

## 1.2 Scopo del prodotto

Lo scopo del prodotto è realizzare un Assistente Virtuale basato su LLM[G], per supportare aziende produttrici di bevande nel fornire informazioni dettagliate e personalizzate sui loro prodotti. Il sistema si rivolge principalmente ai proprietari di locali, consentendo loro di ottenere risposte rapide e precise su caratteristiche, disponibilità e dettagli delle bevande, come se interagissero con uno specialista umano.

## 1.3 Glossario

Per garantire una chiara comprensione della terminologia utilizzata nel documento, è stato predisposto un Glossario[G] in un file dedicato. Questo strumento serve a evitare ambiguità nella definizione dei termini impiegati nell'attività progettuale, offrendo descrizioni precise e condivise.

## 1.4 Riferimenti

### 1.4.1 Riferimenti normativi

- Capitolato C7 LLM[G]: ASSISTENTE VIRTUALE
  `https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C7.pdf`
- Regolamento del progetto didattico
  `https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/PD1.pdf`
- **Norme di Progetto**[G] v.2.0
  `https://code7crusaders.github.io/docs/PB/documentazione_interna/norme_di_progetto.html`

### 1.4.2 Riferimenti informativi

- Slide Corso Ingegneria del software: Analisi dei Requisiti[G]
  `https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/T05.pdf`
- Slide Corso Ingegneria del software: Diagrammi delle classi
  `https://www.math.unipd.it/~rcardin/swea/2023/Diagrammi%20delle%20Classi.pdf`
- Slide Corso Ingegneria del software: Diagrammi dei casi d'uso
  `https://www.math.unipd.it/~rcardin/swea/2022/Diagrammi%20Use%20Case.pdf`
- **Glossario**[G] v.2.0
  `https://code7crusaders.github.io/docs/PB/documentazione_interna/glossario.html`
- Analisi LLM[G]
  `https://code7crusaders.github.io/docs/altri_documenti/analisi_modelli_firmato.html`

- Analisi framework frontend
  `https://code7crusaders.github.io/docs/altri_documenti/analisi_frontend_firmato.html`
- Analisi framework backend
  `https://code7crusaders.github.io/docs/altri_documenti/analisi_framework_backend.html`
- Analisi database Vettoriale
  `https://code7crusaders.github.io/docs/altri_documenti/analisi_dbvettoriale.html`
- **LangChain**[G]
  `https://python.langchain.com/docs/introduction/`
- **OpenAI**
  `https://openai.com/`

# 2 Tecnologie

Questa sezione ha lo scopo di offrire una panoramica delle tecnologie adottate per la realizzazione del sistema software. Vengono analizzati in dettaglio le piattaforme, gli strumenti, i linguaggi di programmazione, i framework e altre risorse tecnologiche utilizzate nel corso dello sviluppo.

## 2.1 Docker

È una piattaforma di containerizzazione leggera che facilita lo sviluppo, il testing e il rilascio delle applicazioni, fornendo un ambiente isolato e riproducibile. Viene utilizzato per creare ambienti di sviluppo uniformi, migliorare la scalabilità delle applicazioni e semplificare la gestione delle risorse.

**Vantaggi:**

- **Portabilità**: I container Docker possono essere eseguiti su qualsiasi piattaforma che supporti Docker, garantendo coerenza tra ambienti di sviluppo, test e produzione.
- **Isolamento delle dipendenze**: Ogni container include tutte le dipendenze necessarie, evitando conflitti tra ambienti.
- **Scalabilità**$^{G}$ facilitata: Permette di scalare i servizi in modo efficiente tramite orchestratori come Kubernetes.
- **Riduzione dei tempi di deploy**: I container vengono avviati rapidamente, accelerando il rilascio delle applicazioni.
- **Maggiore efficienza delle risorse**: I container condividono il kernel del sistema operativo, riducendo l'overhead rispetto alle macchine virtuali.

## 2.2 OpenAI API

L'API di OpenAI fornisce accesso a modelli di intelligenza artificiale avanzati, tra cui modelli di embedding. Un embedding model è un tipo di modello di machine learning che trasforma dati di input, come parole o frasi, in vettori di numeri in uno spazio continuo a bassa dimensione. Questi vettori catturano le caratteristiche semantiche dei dati di input, permettendo di misurare la similarità tra diversi input in modo efficiente.

**Vantaggi:** L'utilizzo di embedding models offre numerosi vantaggi, tra cui:

- **Efficienza**: I vettori di embedding permettono di rappresentare dati complessi in modo compatto e computazionalmente efficiente.
- **Versatilità**: Possono essere utilizzati in una vasta gamma di applicazioni, tra cui il processamento del linguaggio naturale (NLP$^{G}$), la raccomandazione di contenuti e la classificazione dei dati.

**Casi d'uso:** Gli embedding models sono utilizzati in vari casi d'uso, tra cui:

- **Ricerca di documenti**: Migliorano la ricerca di documenti trovando risultati più rilevanti basati sulla similarità semantica.
- **Raccomandazione di contenuti**: Personalizzano le raccomandazioni di contenuti in base alle preferenze dell'utente.
- **Classificazione del testo**: Aiutano nella classificazione automatica di testi in categorie predefinite.

**Impiego del progetto:** Nel progetto, l'API di OpenAI viene utilizzata per convertire il testo in token e generando embedding che rappresentano le caratteristiche semantiche del testo in uno spazio vettoriale.

## 2.3 Linguaggi di programmazione e formato dati

| Nome | Versione | Descrizione | Impiego |
|---|---|---|---|
| Python | 3.0 | Linguaggio di programmazione ad alto livello, dinamico e interpretato | Sviluppo backend, gestione API ed embedding model |
| JavaScript | ES6 | Linguaggio di programmazione interpretato, principalmente utilizzato per lo sviluppo frontend | Sviluppo frontend, interattività delle pagine web, utilizzo di React |
| SQL | - | Linguaggio di programmazione per la gestione e manipolazione di database relazionali | Gestione database, query, manipolazione dati |
| YAML | 1.2 | Formato di serializzazione dati leggibile dall'uomo | Configurazione, script GitHub Actions[G] |
| JSON | - | Formato di interscambio dati leggero e leggibile dall'uomo | Gestione database, scambio dati tra client e server |

Tabella 1: Linguaggi e formati utilizzati

## 2.4 Librerie

| **Python** | | |
|---|---|---|
| **Nome** | **Versione** | **Impiego** |
| `Flask` | 3.1.0 | Framework per applicazioni web in Python. |
| `Flask-Cors` | 5.0.0 | Estensione per Flask per gestire le richieste CORS. |
| `langchain-core` | 0.3.31 | Modulo per la gestione dei documenti in LangChain[G]. |
| `langchain-openai` | 0.3.1 | Integrazione di OpenAI con LangChain[G]. |
| `requests` | 2.32.3 | Libreria per effettuare richieste HTTP. |
| `python-dotenv` | 1.0.1 | Gestione delle variabili d'ambiente da file `.env`. |
| `faiss-cpu` | 1.9.0.post1 | Gestione dei database vettoriali FAISS[G] in LangChain[G]. |
| `numpy` | 2.2.2 | Libreria per il calcolo scientifico e la manipolazione di array. |
| `openai` | 1.60.0 | Libreria per interfacciarsi con l'API di OpenAI. |
| `SQLAlchemy` | 2.0.37 | Toolkit SQL per Python. |
| **JavaScript** | | |
| **Nome** | **Versione** | **Descrizione** |
| `@emotion/react` | 11.14.0 | Libreria per la gestione degli stili CSS-in-JS con supporto a React. |
| `@emotion/styled` | 11.14.0 | API di Emotion per definire componenti con stili CSS-in-JS. |

| Nome | Versione | Impiego |
|---|---|---|
| `@mui/icons-material` | 6.4.8 | Raccolta di icone Material Design per l'uso con MUI. |
| `@mui/material` | 6.4.8 | Libreria di componenti UI basata su Material Design per React. |
| `@reduxjs/toolkit` | 2.6.1 | Toolkit ufficiale per Redux, semplifica la gestione dello stato globale. |
| `@toolpad/core` | 0.13.0 | Framework per costruire applicazioni low-code con React. |
| `react` | 19.0.0 | Libreria per la creazione di interfacce utente basate su componenti. |
| `react-dom` | 19.0.0 | Permette di rendere componenti React nel DOM del browser. |
| `react-markdown` | 10.1.0 | Renderer per il parsing e la visualizzazione di Markdown in React. |
| `react-redux` | 9.2.0 | Binding ufficiale di React per Redux, facilita la connessione ai dati globali. |

Tabella 2: Librerie utilizzate

# 3 Architettura

## 3.1 Introduzione all'architettura

### 3.1.1 Scopo e obbiettivi

La presente sezione ha lo scopo di fornire una visione d'insieme dell'architettura del sistema, evidenziandone i principi guida e le scelte progettuali che ne hanno determinato la struttura. In particolare, si intende:

- Definire il contesto in cui opera il sistema, evidenziando i requisiti funzionali e non funzionali che hanno condotto alla scelta di una specifica architettura.
- Orientare i lettori (sviluppatori, progettisti e stakeholder) verso una comprensione chiara delle componenti principali e delle interazioni che caratterizzano il sistema.
- Porre le basi per la discussione delle scelte di design, evidenziando come queste possano rispondere alle esigenze di scalabilità, sicurezza, manutenibilità e performance.
- Descrivere le motivazioni alla base delle scelte tecnologiche e dei modelli architetturali adottati.

## 3.2 Scelta dell'Architettura del Sistema

La progettazione dell'architettura del sistema ha richiesto un'approfondita analisi delle due principali opzioni architetturali disponibili: **monolitica** e **a microservizi**. La scelta dell'architettura è stata guidata da una serie di fattori, tra cui la natura dell'applicazione, il volume di traffico previsto, i costi di sviluppo e manutenzione e la necessità di scalabilità. In questa sezione verranno analizzati nel dettaglio i pro e i contro di entrambe le soluzioni, per poi motivare la decisione finale.

### 3.2.1 Architettura monolitica

Un'architettura monolitica si basa su un'unica codebase che incorpora tutti i componenti dell'applicazione, tra cui l'interfaccia utente, la logica di business e il livello di accesso ai dati. Questo approccio, tradizionalmente adottato nello sviluppo software, è particolarmente indicato per applicazioni di piccola e media complessità, in cui i costi di separazione dei componenti in unità indipendenti non sono giustificati.

#### Vantaggi

- **Semplicità di sviluppo e gestione**: Un'unica codebase permette di mantenere una visione centralizzata del sistema, semplificando lo sviluppo, il testing e il debugging.
- **Minori costi di infrastruttura**: Non è necessario investire in strumenti di orchestrazione, o gestione della comunicazione tra microservizi.
- **Deployment più semplice**: L'intero sistema viene distribuito come un'unica unità, evitando problemi di coordinamento.
- **Prestazioni migliori per bassi volumi di traffico**: L'assenza di chiamate di rete tra microservizi riduce la latenza.

**Svantaggi**

- **Scalabilità**[G] limitata: Non è possibile scalare singole componenti separatamente.
- **Maggiore impatto degli errori**: Un bug in una parte dell'applicazione può compromettere l'intero sistema.
- **Difficoltà nell'adozione di nuove tecnologie**: L'aggiornamento di singole parti è complesso poiché l'intero stack è integrato.

### 3.2.2 Architettura a microservizi

L'architettura a microservizi suddivide il sistema in componenti indipendenti, ognuno responsabile di una funzionalità specifica. Ogni microservizio comunica con gli altri attraverso API, permettendo un alto grado di indipendenza e flessibilità nello sviluppo.

**Vantaggi**

- **Scalabilità**[G] **orizzontale** : Ogni microservizio può essere scalato indipendentemente.
- **Flessibilità nello sviluppo**: Permette l'adozione di tecnologie diverse per ciascun servizio.
- **Maggiore resilienza**: Un errore in un microservizio non compromette l'intero sistema.
- **Facilità di manutenzione**: È possibile distribuire aggiornamenti senza dover ripubblicare l'intera applicazione.

**Svantaggi**

- **Maggiore complessità gestionale**: L'orchestrazione dei microservizi richiede strumenti avanzati.
- **Comunicazione tra servizi**: Introduce latenza e potenziali colli di bottiglia.
- **Deployment più complesso**: Coordinare il rilascio di più servizi è più oneroso.
- **Costi di sviluppo più elevati**: La frammentazione del sistema richiede maggiore sforzo di progettazione e testing.

### 3.2.3 Architettura esagonale

L'**architettura esagonale**, nota anche come *Ports and Adapters*, è un pattern architetturale l'obiettivo di rendere il software più flessibile, testabile e indipendente dalle tecnologie esterne. Questo approccio enfatizza la separazione tra la logica di business e le interfacce di comunicazione con il mondo esterno. L'architettura esagonale si basa su tre concetti chiave:

- **Isolamento della logica di business**: Il core dell'applicazione è indipendente dai dettagli implementativi esterni.
- **Utilizzo di porte e adattatori**: Le *porte* definiscono le interfacce per la comunicazione tra il core e il mondo esterno, mentre gli *adattatori* implementano queste interfacce per specifiche tecnologie.
- **Sostituibilità delle dipendenze**: È possibile cambiare database, framework web o altre dipendenze senza impattare il core.

L'architettura esagonale può essere rappresentata con tre livelli principali:

1. **Core (Dominio e Logica di Business)**: Contiene le regole fondamentali dell'applicazione.

2. **Porte (Ports)**: Interfacce che definiscono i punti di ingresso e uscita del sistema.
3. **Adattatori (Adapters)**: Implementazioni concrete delle porte per database, servizi esterni e UI.

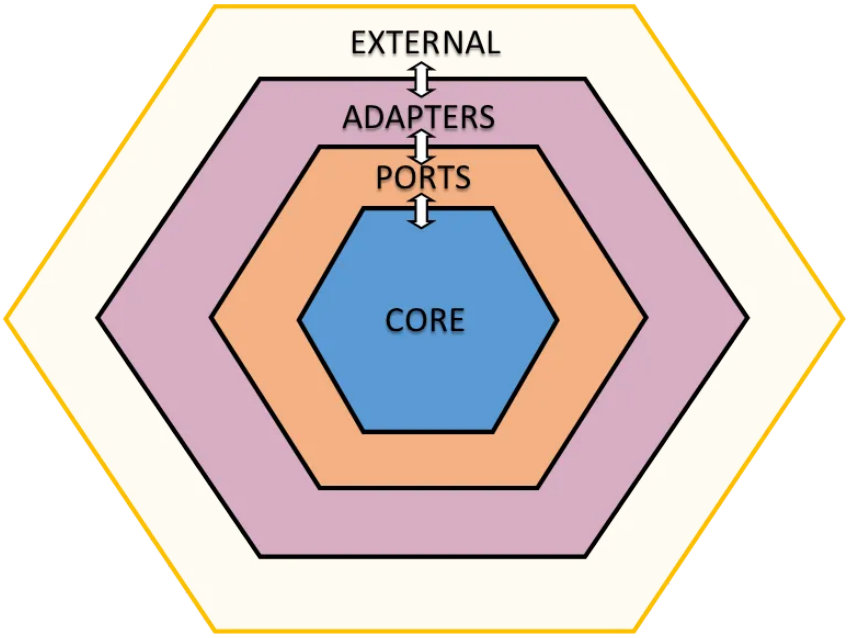


Figura 1: Schema dell'architettura esagonale

Adottare un'architettura esagonale comporta diversi benefici:

- **Maggiore manutenibilità**: Il codice è modulare e separato.
- **Facilità di test**: Il core dell'applicazione può essere testato isolatamente.
- **Indipendenza dalle tecnologie**: Cambiare framework o database ha un impatto minimo.
- **Flessibilità evolutiva**: Permette di trasformare gradualmente il monolite in microservizi.

L'architettura esagonale garantisce modularità e sostenibilità del sistema nel lungo termine, permettendo di scalare senza impattare la stabilità complessiva dell'applicazione.

### 3.2.4 Scelta dell'architettura

Dopo un'analisi approfondita, il team di sviluppo ha deciso di adottare un'**architettura esagonale monolitica**. Questa scelta è stata motivata dai seguenti fattori:

1. **Basso carico di utenti**: L'applicazione è destinata a un contesto B2B con un numero limitato di utenti concorrenti.
2. **Disponibilità non critica 24/7**: Poiché l'applicazione è utilizzata in un contesto aziendale con orari di apertura definiti, non è necessario che il sistema sia sempre online. Questo consente di effettuare manutenzioni anche sull'intero sistema durante i periodi di inattività, senza impatti sull'operatività.
3. **Semplicità di gestione**: La manutenzione di un monolite è più diretta rispetto a un sistema distribuito.
4. **Riduzione dei costi operativi**: L'assenza di strumenti di orchestrazione riduce significativamente i costi di infrastruttura.

5. **Velocità di sviluppo**: Un'unica codebase consente iterazioni rapide senza dipendenze tra servizi separati.

6. **Evoluzione graduale verso microservizi**: Adottando un'architettura **esagonale**, il sistema può essere trasformato gradualmente in microservizi senza riscrivere tutto.

## 3.3 Architettura di Sistema

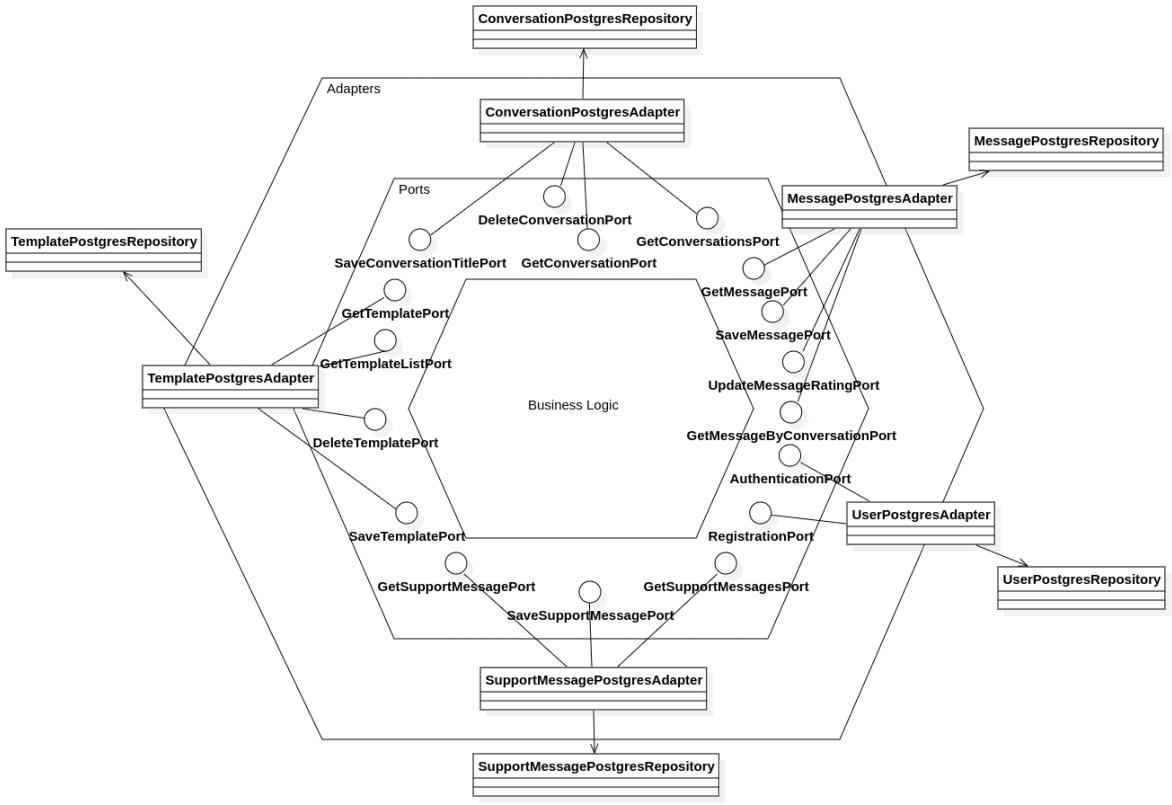


Figura 2: Diagramma delle classi - Architettura Esagonale

Di seguito viene descritta in dettaglio la funzione e l'interazione delle classi della gerarchia proposta, evidenziando il tipo di dato che ciascuna gestisce (entity, model o DTO).

**Repository (usa Entity)**
**Funzione:** Il repository è responsabile della persistenza e del recupero dei dati dal mezzo di archiviazione (ad esempio, un database relazionale o NoSQL).
**Dettagli:**

- **Entity:** Le entity rappresentano il modello del dominio, contenente le informazioni e le regole di business fondamentali.
- **Operazioni tipiche:** CRUD (creazione, lettura, aggiornamento, cancellazione).
- **Astrazione:** Il repository isola il dominio dai dettagli tecnici della persistenza, permettendo al core di rimanere indipendente da framework o tecnologie specifiche.

**Adapter (usa Model)**
**Funzione:** L'adapter funge da mediatore tra il core dell'applicazione e i sistemi esterni o infrastrutture (ad esempio, servizi di terze parti, API, file system).

**Dettagli:**

- **Conversione dei dati:** Trasforma i dati dal formato utilizzato internamente (model) a quello richiesto dal sistema esterno e viceversa.
- **Isolamento delle dipendenze:** Nasconde la complessità delle tecnologie esterne al dominio, conformandosi alle interfacce (port) definite dal core.

**Port (usa Model)**
**Funzione:** I port rappresentano le interfacce o contratti che definiscono il modo in cui il core comunica con il mondo esterno.

**Dettagli:**

- **Definizione del contratto:** Stabiliscono quali operazioni sono disponibili e come devono essere invocate, senza specificare l'implementazione.
- **Indipendenza dal framework:** Consentono al dominio di essere testato e sviluppato senza dipendenze dirette da componenti esterni.

**Service (usa Model)**
**Funzione:** I service aggregano e orchestrano la logica di business complessa, spesso condivisa tra più casi d'uso.

**Dettagli:**

- **Coordinamento delle operazioni:** Chiamano i repository per accedere ai dati, invocano i port per interagire con sistemi esterni e applicano le regole di business.
- **Modularità:** Incapsulano comportamenti riutilizzabili, mantenendo il core dell'applicazione pulito e focalizzato sulle regole di business.

**Usecase (usa Model)**
**Funzione:** Gli usecase rappresentano le singole operazioni o workflow che l'applicazione offre; sono casi d'uso specifici del dominio.

**Dettagli:**

- **Incapsulamento del flusso di lavoro:** Ogni usecase definisce un'intera operazione (ad es. "crea ordine", "effettua pagamento"), coordinando il service e altre componenti necessarie per completarla.
- **Gestione del modello:** Utilizzano il model per rappresentare i dati che vengono manipolati durante il caso d'uso, garantendo la coerenza con le regole di business.

**Controller (usa DTO)**
**Funzione:** Il controller è il punto di ingresso per le richieste esterne (tipicamente interfacce web, API REST[G], interfacce utente) e si occupa di tradurle nel linguaggio comprensibile dal dominio.

**Dettagli:**

- **Utilizzo dei DTO:** I Data Transfer Object (DTO) sono strutture dati leggere che trasportano le informazioni tra il client e il server, evitando di esporre direttamente il modello di dominio o le entity.
- **Validazione e mapping:** Il controller riceve i dati in formato DTO, li valida e li converte in input per i usecase; allo stesso modo, trasforma i risultati (model) in DTO da restituire all'utente.

**Flusso Complessivo dei Dati**

1. **Ingresso:** Il controller riceve una richiesta (es. HTTP) con i dati in formato DTO.
2. **Esecuzione del Usecase:** Il controller passa questi dati a un usecase, che rappresenta un'operazione specifica del dominio.
3. **Business Logic:** Il usecase, eventualmente tramite un service, applica la logica di business usando il model e interagendo con i port.
4. **Interazione Esterna:** Se necessario, un adapter viene utilizzato per comunicare con sistemi esterni (ad es. per salvare dati), passando attraverso il port che definisce il contratto.
5. **Persistenza:** Il repository si occupa della persistenza, lavorando direttamente con le entity che rappresentano i dati fondamentali.
6. **Risposta:** Una volta completata l'operazione, il risultato viene ritrasformato (tramite mapping) in un DTO e restituito al client attraverso il controller.

**Vantaggi di questa Architettura**

- **Isolamento del dominio:** Il core dell'applicazione è isolato da dettagli tecnici e variazioni infrastrutturali, facilitando test, manutenzione e scalabilità.
- **Flessibilità:** I port e gli adapter permettono di sostituire facilmente componenti esterni (ad es. cambiare il database o il sistema di invio email) senza impattare la logica di business.
- **Chiarezza e Separazione dei Compiti:** La divisione in repository, adapter, port, service, usecase e controller aiuta a mantenere un'architettura modulare, in cui ogni componente ha un compito ben definito.

## 3.4 Moduli

### 3.4.1 Chat Controller

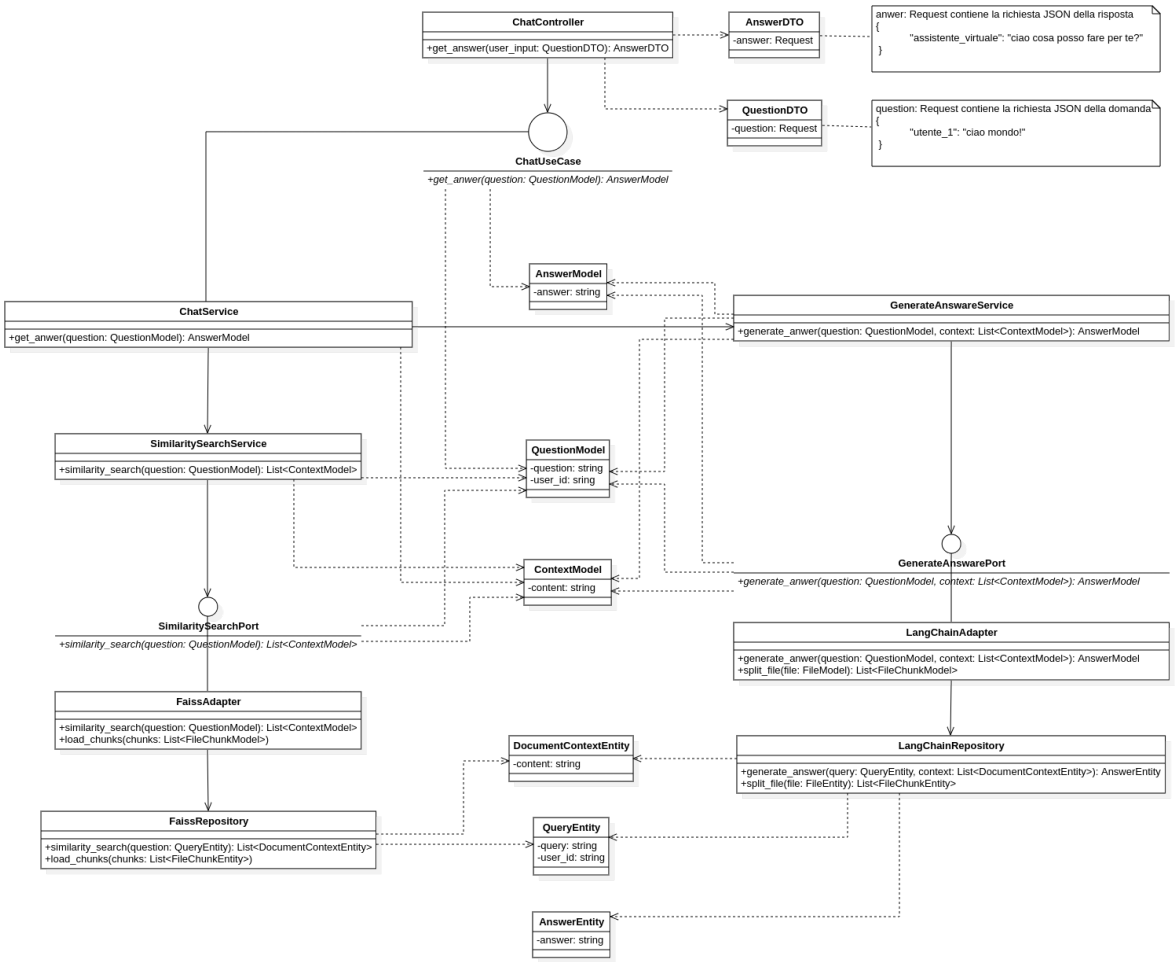


Figura 3: Diagramma delle classi - Chat Controller

**1. Data Transfer Objects (DTO)**

- **AnswerDTO & QuestionDTO:**
  - Questi oggetti sono usati per trasferire i dati tra le varie componenti (ad esempio, tra il controller e il caso d'uso).
  - AnswerDTO incapsula la risposta generata.
  - QuestionDTO contiene informazioni relative all'utente (identificato da un intero) e alla domanda posta.

**2. Modelli ed Entità**

- **Modelli (Models):**
  - Rappresentano le strutture dati a livello di dominio, per esempio:
    - *QuestionModel*: Tiene traccia dell'ID utente e del testo della domanda.
    - *AnswerModel*: Incapsula la risposta generata.

    * *ContextModel*: Utilizzato per rappresentare il contesto (ad esempio, contenuti estratti da documenti) che verrà usato per generare una risposta.

- **Entità:**
  - Gli oggetti di tipo entità, come DocumentContextEntity, QueryEntity e AnswerEntity, rappresentano versioni "di basso livello" degli stessi concetti, ma con logiche e metodi per accedere ai dati (ad esempio, `get_content()` o `get_query()`).
  - Queste entità sono utili per trasformare i dati dai modelli alle strutture usate nelle operazioni di business.

**3. Repositories e Interazione con il Vector Store**

- **FaissRepository:**
  - Questo componente interagisce direttamente con un vector store basato su FAISS[G].
  - `similarity_search(query)`: Riceve una QueryEntity, esegue una ricerca di similarità sul vector store (limitata ad un certo numero di risultati, ad esempio 4) e trasforma i documenti trovati in oggetti DocumentContextEntity.
  - `load_chunks(chunks)`: Consente di caricare "chunk" di testo (rappresentati come FileChunkEntity) nel vector store. Per ciascun chunk viene creato un oggetto Document con metadati, successivamente il vector store viene salvato in maniera persistente.

**4. Integrazione con LangChain**

- **LangChainRepository:**
  - Utilizza le funzionalità di LangChain[G] per la generazione di risposte e per la suddivisione dei file.
  - `generate_answer(query, contexts, prompt_template)`:
    1. Prepara una lista di documenti (tramite Document di LangChain[G]) a partire dai contesti ricevuti.
    2. Recupera la "memoria" utente per mantenere la storia delle conversazioni, la quali vengono trimmate se troppo lunghe per non superare un limite di token.
    3. Costruisce dinamicamente un prompt (usando ChatPromptTemplate) che include istruzioni di sistema, la storia della conversazione, la domanda attuale e il contesto.
    4. Invoca una catena (chain) per ottenere la risposta dall'LLM[G].
    5. Salva l'interazione nella memoria utente e restituisce la risposta incapsulata in un AnswerEntity.
  - `split_file(file)`: Suddivide il contenuto di un file in "chunk" di dimensioni definite (ad es. 2500 caratteri) utilizzando lo RecursiveCharacterTextSplitter. Se il contenuto è in bytes, viene decodificato in stringa. Il risultato è una lista di oggetti FileChunkEntity.

**5. Adattatori (Adapters)**

- **FaissAdapter:**
  - Implementa l'interfaccia SimilaritySearchPort e AddChunksPort.
  - Il metodo `similarity_search()` trasforma un QuestionModel in una QueryEntity.
  - Converte i risultati del repository in istanze di ContextModel.

- **LangChainAdapter:**
  - Implementa le interfacce GenerateAnswerPort e SplitFilePort.
  - `generate_answer()` converte il QuestionModel e la lista di ContextModel in entità adatte alla generazione della risposta.
  - `split_file()` trasforma un FileModel in un FileEntity e poi converte i chunk ottenuti in oggetti FileChunkModel.

**6. Interfacce (Ports)**

- Definiscono i contratti per le funzionalità principali:
  - **SimilaritySearchPort:** Definisce il metodo per eseguire la ricerca di similarità dati un QuestionModel.
  - **GenerateAnswerPort:** Definisce il metodo per generare una risposta basata su domanda, contesto e prompt template.

**7. Servizi**

- **SimilaritySearchService:** Gestisce la ricerca di similarità per un QuestionModel.
- **GenerateAnswerService:** Invoca la generazione della risposta e gestisce la propagazione degli errori.

**8. Use Case e ChatService**

- **ChatUseCase (interfaccia astratta) e ChatService (implementazione):**
  - `get_answer(question_model)` coordina il processo di recupero del contesto e generazione della risposta.

**9. Controller**

- **ChatController:**
  - Funziona da interfaccia verso l'esterno (ad esempio, per una API REST[G]).
  - `get_answer(user_input)`:
    1. Converte il QuestionDTO in un QuestionModel.
    2. Chiama il caso d'uso ChatUseCase per ottenere la risposta.
    3. Converte il risultato (AnswerModel) in un AnswerDTO da restituire al chiamante.

### 3.4.2 Add File Controller

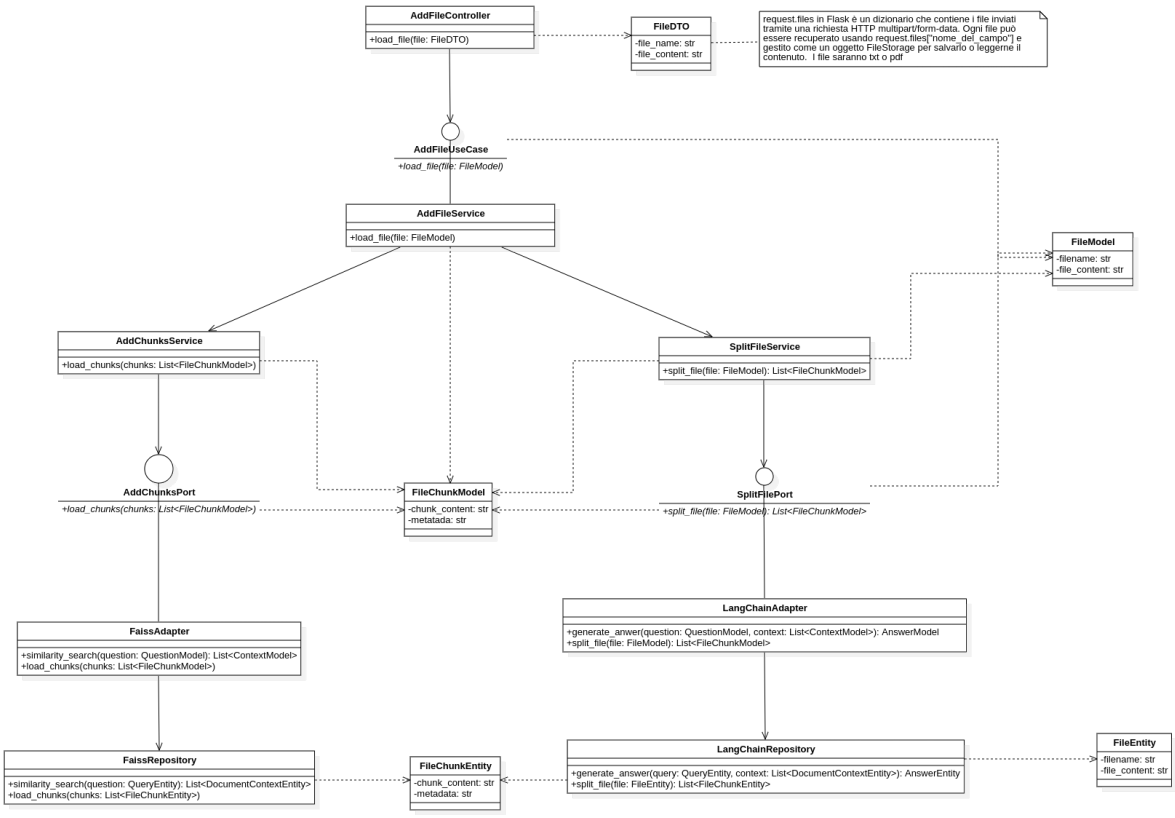


Figura 4: Diagramma delle classi - Add File Controller

Questo modulo implementa una struttura modulare per gestire il caricamento, il processamento e la ricerca di file (e dei loro contenuti). Di seguito viene fornita una spiegazione dettagliata dei componenti principali:

**1. Data Transfer Object (DTO) e Modelli di Dati**

- **FileDTO:**
  - Funziona da oggetto di trasferimento dati per il file.
  - **Attributi:**
    - file_name: Nome del file.
    - file_content: Contenuto del file.
  - **Metodi:**
    - get_file_name(): Ritorna il nome del file.
    - get_file_content(): Ritorna il contenuto del file.
- **FileModel:**
  - Rappresenta il file nel dominio interno all'applicazione.
  - **Attributi:**
    - filename: Nome del file.
    - file_content: Contenuto del file.

  - **Metodi:**
    - get_filename(): Ritorna il nome del file.
    - get_file_content(): Ritorna il contenuto del file.

- **FileChunkModel:**
  - Modella un frammento (chunk) di file.
  - **Attributi:**
    - chunk_content: Il contenuto del frammento.
    - metadata: Metadati associati al frammento (ad es. informazioni sul contesto o origine).
  - **Metodi:**
    - get_chunk_content(): Ritorna il contenuto del chunk.
    - get_metadata(): Ritorna i metadati.

- **FileEntity e FileChunkEntity:**
  - **FileEntity:**
    - Attributi: metadata e file_content.
    - Metodi: get_metadata() e get_file_content().
  - **FileChunkEntity:**
    - Attributi: chunk_content e metadata.
    - Metodi: get_chunk_content() e get_metadata().

**2. Controller e Use Case**

- **AddFileController:**
  - **Ruolo:** Gestisce la richiesta di aggiunta di un file alla base dati.
  - **Flusso:**
    - Riceve un oggetto FileDTO.
    - Converte il DTO in un FileModel (adattando il formato per il dominio interno).
    - Invoca il metodo load_file del use case associato (AddFileUseCase).
  - **Gestione Errori:** Nel blocco try/except, eventuali errori vengono rilanciati.

- **AddFileUseCase (astratto):**
  - **Scopo:** Definisce l'interfaccia per il caso d'uso di aggiunta file.
  - **Metodo astratto:** load_file(file: FileModel) che deve essere implementato da una classe concreta.

**3. Servizi**

- **AddFileService:**
  - **Ruolo:** Gestisce la logica di business per il caricamento del file e la gestione dei suoi frammenti.
  - **Metodi principali:**
    - load_file(file: FileModel):
      - Effettua lo split del file in frammenti (chiamando il metodo split_file).
      - Carica i frammenti tramite load_chunks.

        * split_file(file: FileModel) - list[FileChunkModel]:
            · Utilizza il servizio SplitFileService per dividere il file in chunk.
        * load_chunks(chunks: list[FileChunkModel]):
            · Inoltra i chunk al servizio AddChunksService.

- **AddChunksService:**
  - **Ruolo:** Si occupa del caricamento dei frammenti nel sistema di repository.
  - **Metodo:**
    * load_chunks(chunks: list[FileChunkModel]):
        · Invoca il metodo load_chunks del port AddChunksPort.

- **SplitFileService:**
  - **Ruolo:** Incapsula la logica per la divisione di un file in frammenti.
  - **Metodo:**
    * split_file(file: FileModel) → list[FileChunkModel]:
        · Delegato al port SplitFilePort.

**4. Port e Interfacce Astratte**

- **AddChunksPort:**
  - **Scopo:** Definisce il contratto per il caricamento dei chunk in un repository (es. FAISS[G]).
  - **Metodo astratto:** load_chunks(chunks: list[FileChunkModel]).

- **SplitFilePort:**
  - **Scopo:** Definisce il contratto per la divisione di un file in frammenti.
  - **Metodo astratto:** split_file(file: FileModel) → list[FileChunkModel].

**5. Adapter e Repository**

- **FaissAdapter:**
  - **Implementa:** SimilaritySearchPort e AddChunksPort.
  - **Funzionalità:**
    * Similarity Search:
        · Metodo similarity_search(question_model: QuestionModel):
        · Verifica che la domanda non sia vuota.
        · Crea un'entità query (QueryEntity).
        · Richiama il metodo similarity_search del repository FAISS[G].
        · Trasforma i risultati in oggetti ContextModel.
    * Load Chunks:
        · Metodo load_chunks(chunks: list[FileChunkModel]):
        · Converte i modelli in entità (FileChunkEntity) e li passa al repository FAISS[G].
        · Salva lo stato aggiornato del vector store.

### 3.4.3 Conversation

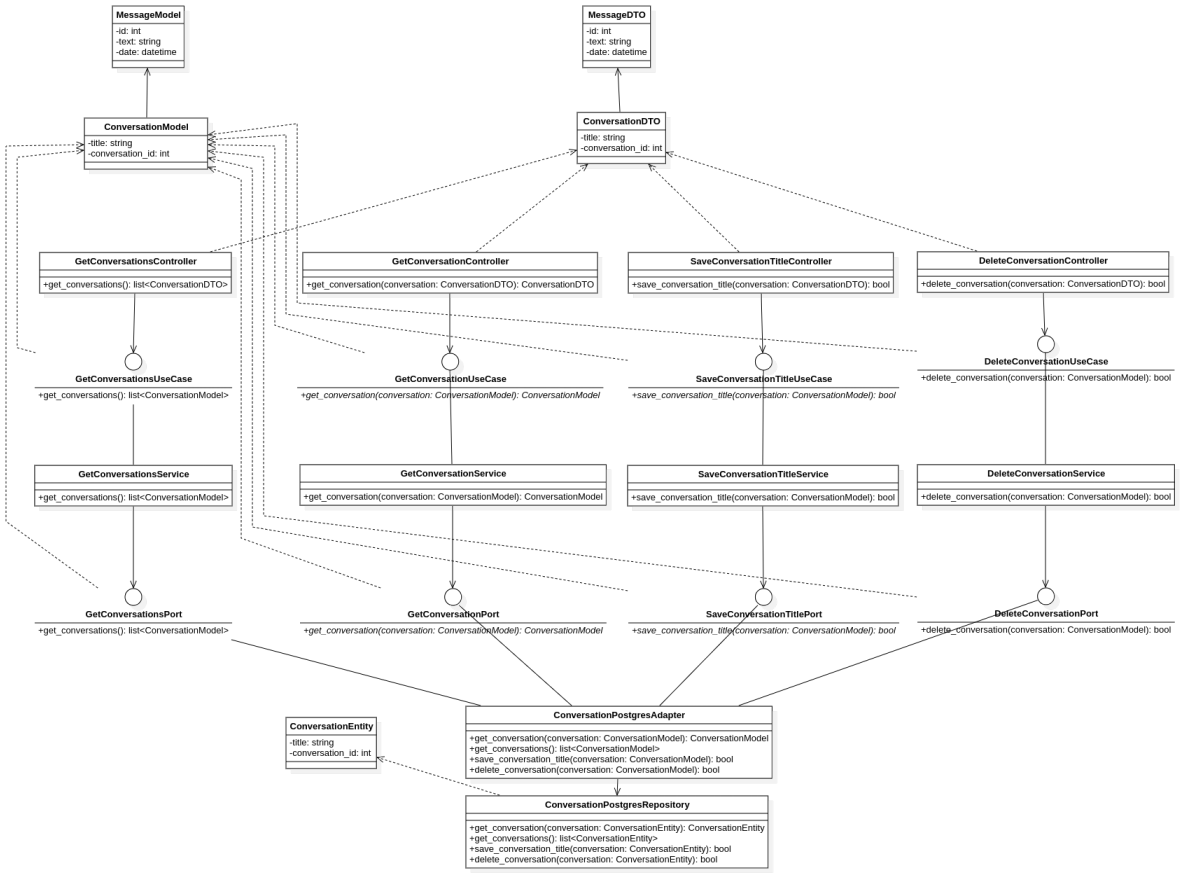


Figura 5: Diagramma delle classi - Conversation

Il modulo implementa un'architettura a strati (simile alla Clean Architecture) per la gestione delle conversazioni e dei messaggi in un'applicazione. Di seguito viene descritta in dettaglio la struttura e il funzionamento di ciascuna parte.

1. **Data Transfer Objects (DTO)**
   - **MessageDTO & ConversationDTO:**
     - Questi oggetti fungono da oggetti di trasferimento dati tra i vari livelli dell'applicazione (ad esempio, tra controller e use case).
     - *MessageDTO*: Contiene attributi come id, text, is_bot, conversation_id, rating e created_at.
     - *ConversationDTO*: Contiene attributi come id, title e user_id e metodi getter per accedervi.
2. **Modelli**
   - **MessageModel & ConversationModel:**
     - Questi modelli rappresentano la struttura dei dati a livello di dominio, simili ai DTO ma utilizzati internamente (per la logica di business).
     - *MessageModel*: Replicano gli attributi di MessageDTO e forniscono metodi getter per recuperarli.
     - *ConversationModel*: Replicano gli attributi di ConversationDTO e forniscono metodi getter per recuperarli.

**3. Controller**

- I controller sono responsabili della gestione delle richieste e della delega dell'elaborazione ai use case. In questo codice, sono presenti quattro controller:
  - **GetConversationController:**
    - Riceve un oggetto ConversationDTO, lo converte in un ConversationModel e lo passa al use case `get_conversation`.
    - Una volta ottenuto il risultato dal use case, lo trasforma in un nuovo ConversationDTO e lo restituisce.
  - **GetConversationsController:**
    - Lavora con una lista di conversazioni, convertendo ogni ConversationDTO in un ConversationModel.
    - Richiama il use case per ottenere una lista di conversazioni e mappa ciascun elemento della lista in un ConversationDTO.
  - **SaveConversationTitleController:**
    - Riceve un ConversationDTO contenente il titolo e altri dati.
    - Converte il DTO in un model e invoca il use case `save_conversation_title` per salvare il titolo nel database.
    - Restituisce l'ID della conversazione salvata.
  - **DeleteConversationController:**
    - Riceve un ConversationDTO contenente il titolo e altri dati.
    - Converte il DTO in un model e invoca il use case `delete_conversation` per eliminare la conversazione.
    - Restituisce True se l'operazione va a buon fine, False altrimenti.

**4. Use Case e Interfacce Astratte**

- I use case definiscono la logica applicativa e le operazioni principali. Sono implementati tramite classi astratte che fungono da interfacce:
  - **GetConversationUseCase:** Definisce il metodo astratto `get_conversation`, che restituisce un altro ConversationModel (recuperato dal database).
  - **GetConversationsUseCase:** Definisce il metodo astratto `get_conversations` per ottenere una lista di conversazioni.
  - **SaveConversationTitleUseCase:** Definisce il metodo astratto `save_conversation_title` per salvare il titolo di una conversazione e restituire l'ID aggiornato.
  - **DeleteConversationUseCase:** Definisce il metodo astratto `delete_conversation` per eliminare una conversazione.

**5. Service**

- I service implementano concretamente i use case e delegano l'accesso ai dati ai cosiddetti *port* (interfacce di adattamento):
  - **GetConversationService:** Implementa `get_conversation` invocando il metodo `get_conversation` del relativo port.

- **GetConversationsService:** Implementa `get_conversations` invocando il metodo `get_conversations` del port.
- **SaveConversationTitleService:** Implementa `save_conversation_title` invocando il metodo `save_conversation_title` del relativo port.
- **DeleteConversationService:** Implementa `delete_conversation` invocando il metodo `delete_conversatio` del relativo port.

**6. Ports**

- I port sono interfacce che definiscono i metodi per l'interazione con il livello di persistenza (repository). Sono definiti come classi astratte:
  - **GetConversationPort:** Specifica il metodo `get_conversation` per recuperare una conversazione.
  - **GetConversationsPort:** Specifica il metodo `get_conversations` per recuperare tutte le conversazioni relative a un utente.
  - **SaveConversationTitlePort:** Specifica il metodo `save_conversation_title` per salvare il titolo di una conversazione.
  - **DeleteConversationPort:** Specifica il metodo `delete_conversation` per eliminare una conversazione.

**7. Adapter e Repository**

- **Adapter: ConversationPostgresAdapter:**
  - Questa classe implementa i port (`GetConversationPort`, `GetConversationsPort`, `SaveConversationTitlePort`, `DeleteConversationPort`) e funge da intermediario tra il livello service e il repository.
- **Repository: ConversationPostgresRepository:**
  - Si occupa della comunicazione diretta con il database PostgreSQL[G] tramite il modulo `psycopg2`.
  - Fornisce metodi per la connessione al database, l'esecuzione di query per ottenere, salvare ed eliminare conversazioni.

**8. Flusso Complessivo**

- Ricezione della richiesta: Un controller riceve un DTO dalla parte esterna.
- Conversione e delega al use case: Il controller trasforma il DTO in un Model e lo passa al use case.
- Accesso ai dati tramite il Port: Il Service invoca il metodo del Port.
- Operazione sul database: L'Adapter chiama il repository per eseguire query.
- Restituzione della risposta: Il repository restituisce i dati all'Adapter, che li trasforma e li passa al Controller per la risposta finale.

### 3.4.4 Message

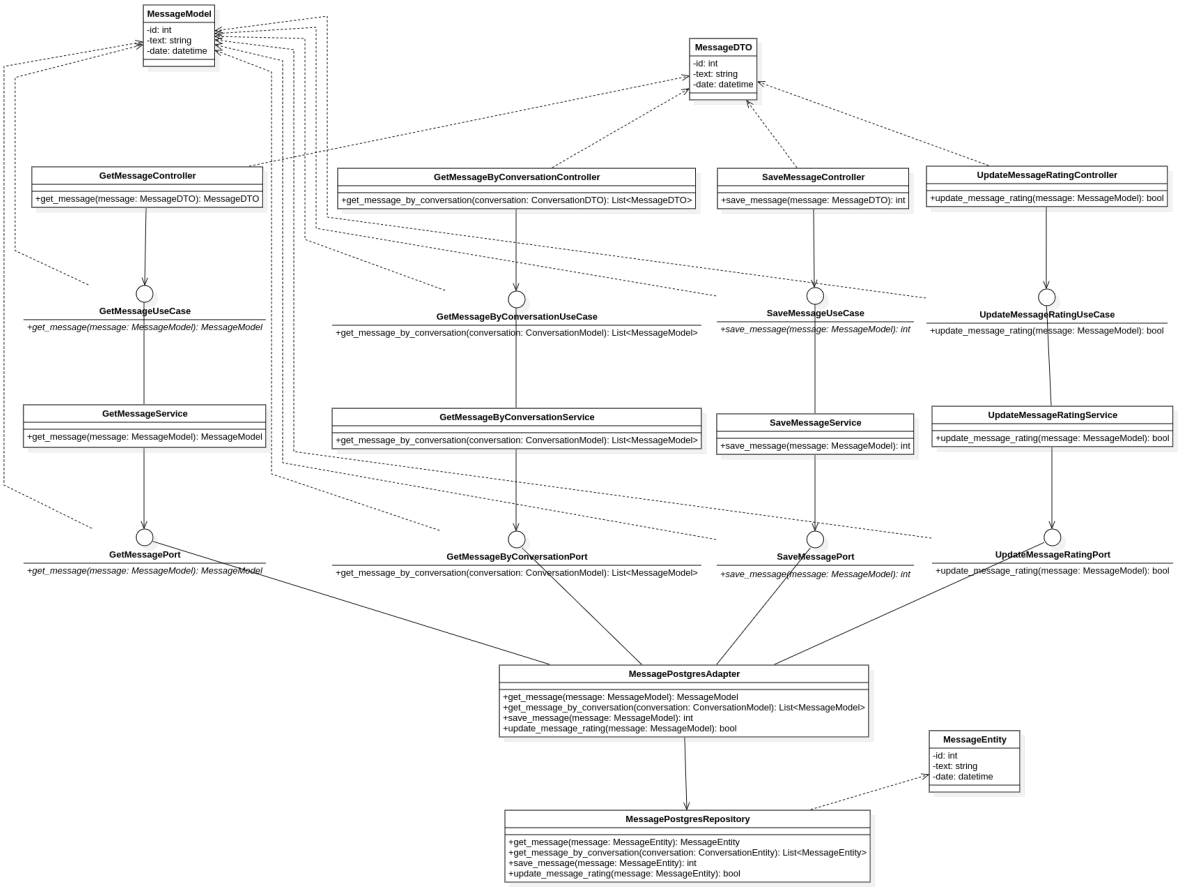


Figura 6: Diagramma delle classi - Message

Il modulo implementa un'architettura a più livelli per la gestione dei messaggi, separando le responsabilità in vari componenti (DTO, modelli, entità, controller, use case, servizi, porte e repository) per garantire una maggiore manutenibilità, testabilità ed estendibilità dell'applicazione. Di seguito una descrizione dettagliata dei vari componenti e del loro funzionamento:

**1. Data Transfer Objects (DTO), Modelli ed Entità**

- **MessageDTO:**
  - È un oggetto di trasferimento dati (DTO) che incapsula le informazioni di un messaggio.
  - **Campi:** `id`, `text`, `is_bot`, `conversation_id`, `rating`, `created_at`.
  - **Metodi Getter:** `get_id()`, `get_text()`, ecc.
  - **Utilizzo:** Facilita lo scambio dei dati tra il livello di presentazione e i livelli di business/logica.
- **MessageModel:**
  - Rappresenta il modello di dominio per il messaggio.
  - Ha gli stessi attributi e metodi del DTO, ma viene utilizzato all'interno della logica applicativa per eseguire operazioni, trasformazioni o validazioni.
- **MessageEntity:**

- È la rappresentazione della struttura dati a livello di persistenza (database).
- Incapsula gli stessi dati del DTO e Model, ma viene utilizzato dal repository per interagire con il database.

**2. Controller**

- **GetMessageController:**
  - Funzione: Recupera un messaggio dato il suo ID.
  - Processo:
    1. Converte il `MessageDTO` ricevuto in un `MessageModel`.
    2. Chiama il metodo `get_message` del relativo use case.
    3. Se il messaggio viene trovato, viene convertito nuovamente in un `MessageDTO`.
  - Error Handling: Gestisce eventuali eccezioni rilanciate.
- **GetMessagesByConversationController:**
  - Funzione: Recupera tutti i messaggi relativi ad una specifica conversazione.
  - Processo:
    1. Converte il DTO della conversazione in un modello.
    2. Invoca il metodo `get_messages_by_conversation` del use case.
    3. Crea un nuovo `MessageDTO` per ogni modello restituito e restituisce la lista.
  - Error Handling: Rilancia eventuali eccezioni.
- **SaveMessageController:**
  - Funzione: Salva un nuovo messaggio nel database.
  - Processo:
    1. Converte il `MessageDTO` in un `MessageModel`.
    2. Chiama il metodo `save_message` del use case.
    3. Restituisce l'ID del messaggio salvato.
  - Error Handling: Gestisce le eccezioni.
- **UpdateMessageRatingController:**
  - Funzione: Aggiorna il rating di un messaggio.
  - Processo:
    1. Converte il `MessageDTO` in un `MessageModel`.
    2. Chiama il metodo `update_message_rating` del use case.
    3. Restituisce True se l'operazione va a buon fine, False altrimenti.
  - Error Handling: Gestisce le eccezioni.

**3. Use Case (Interfacce astratte e Implementazioni di servizio)**

- **GetMessageUseCase:**
  - Definisce il metodo `get_message` che, dato un `MessageModel`, restituisce il messaggio corrispondente.

- **GetMessagesByConversationUseCase:**
  - Definisce il metodo `get_messages_by_conversation` per ottenere una lista di messaggi di una conversazione.
- **SaveMessageUseCase:**
  - Definisce il metodo `save_message` per salvare un messaggio e restituire l'ID.
- **UpdateMessageRatingUseCase:**
  - Definisce il metodo `update_message_rating` per aggiornare il rating di un messaggio.

**4. Porte (Interfaces) e Adapter**

- **GetMessagePort, GetMessagesByConversationPort, SaveMessagePort, UpdateMessageRatingPort:**
  - Ogni interfaccia espone il metodo necessario per eseguire l'operazione di recupero o salvataggio.
  - Queste interfacce sono implementate da un adattatore concreto.
- **MessagePostgresAdapter:**
  - **Funzioni principali:**
    1. `get_message`: Converte un `MessageModel` in `MessageEntity` e richiama il metodo `get_message` del repository.
    2. `get_messages_by_conversation`: Converte l'input in una `MessageEntity` e invoca il repository per ottenere la lista di messaggi.
    3. `save_message`: Converte il `MessageModel` in un `MessageEntity`, invoca il repository per il salvataggio e restituisce l'ID.
    4. `update_message_rating`: Converte il `MessageModel` in un `MessageEntity`, invoca il repository per il salvataggio e restituisce un booleano.

**5. Repository**

- **MessagePostgresRepository:**
  - **__init__**: Inizializza il repository con una configurazione del database.
  - **__connect**: Crea una connessione al database.
  - `get_message`: Esegue una query per recuperare un messaggio in base all'ID.
  - `get_messages_by_conversation`: Esegue una query per ottenere tutti i messaggi di una conversazione.
  - `save_message`: Inserisce un messaggio nel database e restituisce l'ID.
  - `update_message_rating`: Aggiorna il rating di un messaggio nel database e restituisce un booleano.
  - `delete_message`: (opzionale) Cancella un messaggio dato l'ID.

**6. Integrazione e Flusso Complessivo**

- Il flusso per salvare o recuperare un messaggio è il seguente:
  1. Il controller riceve un `MessageDTO`.
  2. Converte il DTO in un `MessageModel`.
  3. Chiama il metodo appropriato del use case.
  4. Il use case delega la richiesta alla porta (ad esempio, `MessagePostgresAdapter`).
  5. La porta chiama il repository per eseguire l'operazione sul database.
  6. I dati vengono ritrasformati fino a essere restituiti come `MessageDTO`.

### 3.4.5 User

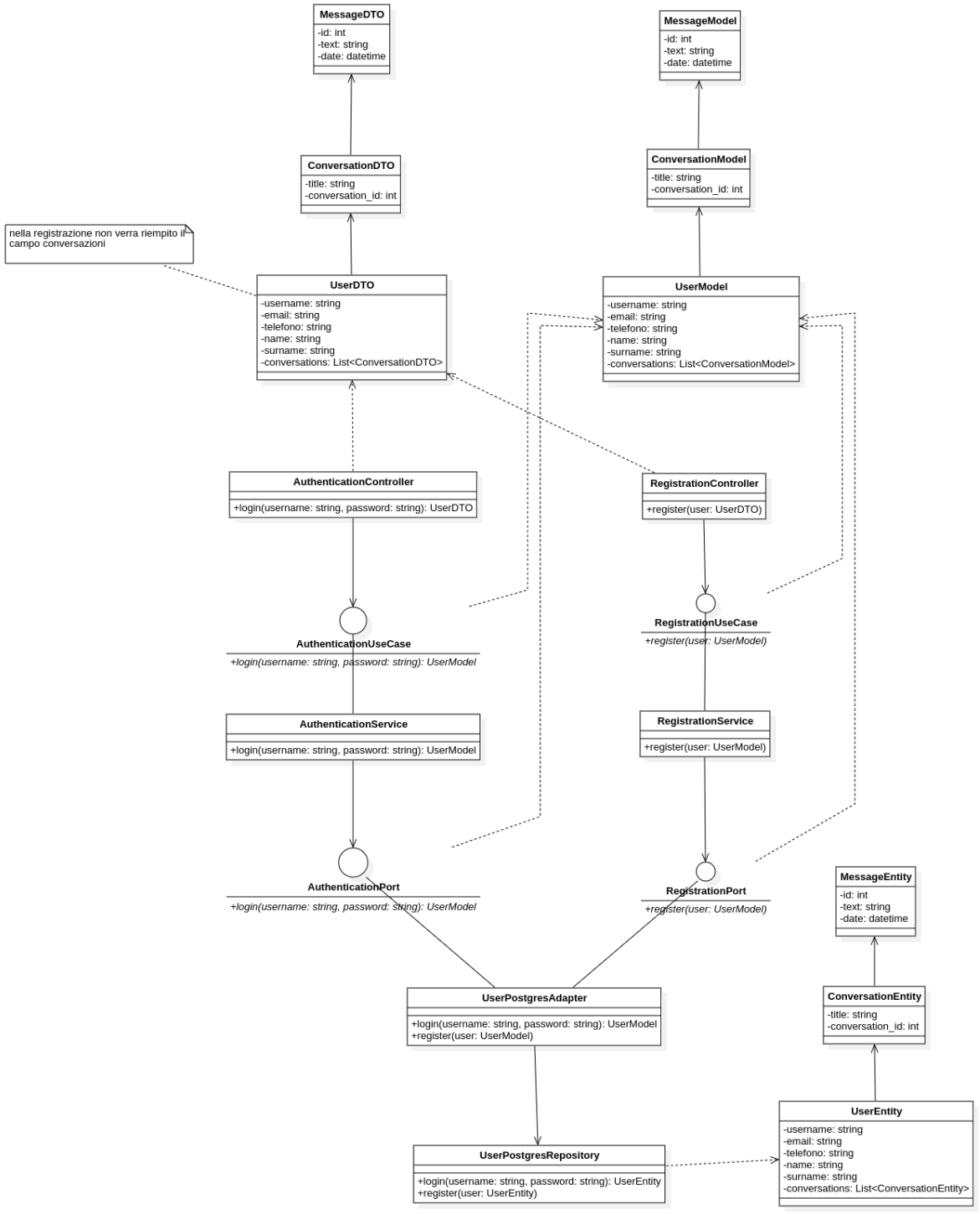


Figura 7: Diagramma delle classi - User

## 1. Data Transfer Objects (DTO)

- **MessageDTO:**
  - Rappresenta un messaggio in una conversazione.
  - Attributi: `id`, `text`, `is_bot`, `conversation_id`, `rating`, `created_at`.

  - Ha metodi getter per recuperare i valori.

- **ConversationDTO:**
  - Rappresenta una conversazione.
  - Attributi: `id`, `title`, `user_id`.
  - Ha metodi getter.

- **UserDTO:**
  - Rappresenta un utente.
  - Attributi: `id`, `username`, `password`, `email`, `phone`, `first_name`, `last_name`, `is_admin`.
  - Ha metodi getter e `set_password` per modificare la password.

**2. Modelli e Entità**

- **MessageModel:**
  - Stessi attributi del `MessageDTO`.
  - Metodi getter per ottenere i valori.

- **ConversationModel:**
  - Stessi attributi di `ConversationDTO`.

- **UserModel:**
  - Stessi attributi di `UserDTO`.
  - Contiene un metodo `set_password` per aggiornare la password.

**3. Entità**

- **MessageEntity, ConversationEntity, UserEntity:**
  - Stessi attributi delle corrispondenti classi Model e DTO.

**4. Controller**

- **AuthenticationController:**
  - Metodo `login(user_dto: UserDTO) -> UserDTO`.
  - Converte `UserDTO` in `UserModel`.
  - Chiama `authentication_use_case.login(user_model)`.
  - Restituisce un `UserDTO` basato sull'output.

- **RegistrationController:**
  - Metodo `register(user_dto: UserDTO) -> bool`.
  - Converte `UserDTO` in `UserModel`.
  - Chiama `registration_use_case.register(user_model)`.
  - Restituisce `True` o `False`.

**5. Use Case**

- **AuthenticationUseCase:**
  - Metodo `login(user_model: UserModel) -> UserModel` (non implementato).
- **RegistrationUseCase:**
  - Metodo `register(user_model: UserModel) -> bool` (non implementato).

**6. Servizi**

- **AuthenticationService:**
  - Implementa `login(user_model: UserModel) -> UserModel`.
  - Recupera l'utente dal database tramite `authentication_port.get_user_for_authentication(user_model)`.
  - Controlla la validità della password con `bcrypt.check_password_hash`.
  - Se tutto è corretto, restituisce l'utente autenticato.
- **RegistrationService:**
  - Implementa `register(user_model: UserModel) -> bool`.
  - Valida i dati dell'utente.
  - Hash della password con `bcrypt.generate_password_hash`.
  - Registra l'utente nel database tramite `registration_port.register(user_model)`.

**7. Ports**

- **AuthenticationPort:**
  - Definisce `get_user_for_authentication(user_model: UserModel) -> UserModel`.
- **RegistrationPort:**
  - Definisce `register(user_model: UserModel) -> bool`.

**8. Repository**

- **UserPostgresRepository:**
  - Usa `psycopg2` per connettersi a PostgreSQL[G].
  - `register(user_model: UserEntity) -> bool`.
  - Esegue `INSERT INTO Users (...) VALUES (...)` per salvare l'utente.
  - `get_user_by_email(email: str) -> bool`.
  - Controlla se un utente con una determinata email esiste nel database.
  - `get_user_by_username(username: str) -> bool`.
  - Controlla se un utente con un determinato username esiste.
  - `get_user_for_authentication(user: UserEntity) -> UserEntity`.
  - Recupera un utente per il login in base all'username.

**9. Adattatori (Adapters)**

- **UserPostgresAdapter:**
  - Implementa le interfacce `RegistrationPort`, `ValidationPort`, `AuthenticationPort`.
  - Converte `UserModel` in `UserEntity` prima di chiamare il repository.
  - Fornisce i metodi:
    * `register(user_model: UserModel) -> bool`.
    * `get_user_by_email(email: str) -> bool`.
    * `get_user_by_username(username: str) -> bool`.
    * `get_user_for_authentication(user_model: UserModel) -> UserModel`.

### 3.4.6 Support Message

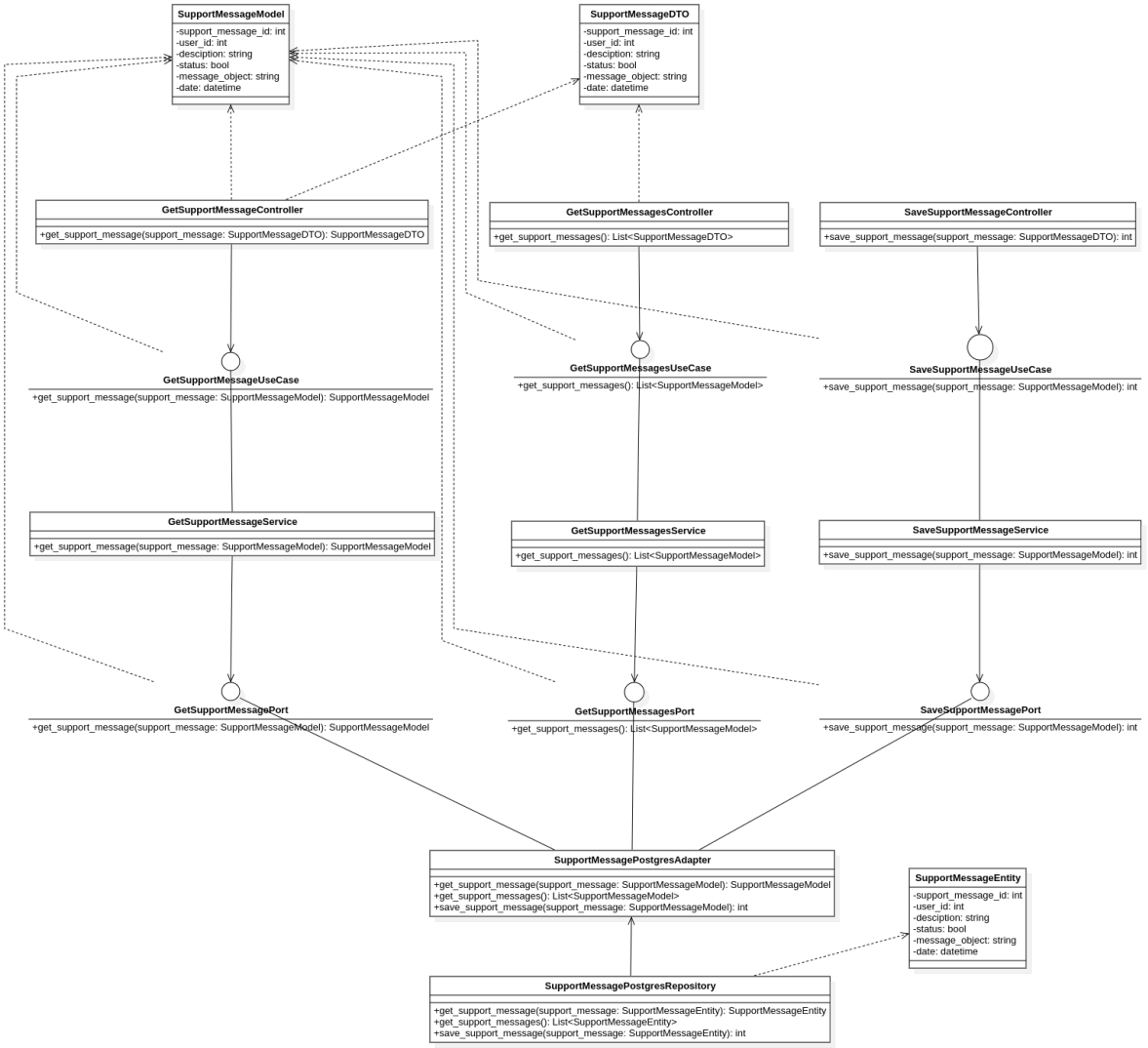


Figura 8: Diagramma delle classi - Support Message

**1. Data Transfer Objects (DTO)**

- **SupportMessageDTO:**

- *Scopo:* Funziona da Data Transfer Object, utilizzato per trasportare i dati tra i vari strati dell'applicazione (ad esempio, dal controller al service o viceversa).
- *Attributi:* id, user_id, description, status, subject, created_at.
- *Metodi:* Getter per ciascun attributo (ad esempio, get_id(), get_user_id(), ecc.).
- *Utilizzo:* Incapsula i dati di un messaggio di supporto e li trasporta tra le componenti dell'applicazione senza esporre direttamente le strutture interne.

**2. Modelli ed Entità**

- **SupportMessageModel:**
  - *Scopo:* Rappresenta il modello di dominio del messaggio di supporto.
  - *Caratteristiche:* Struttura simile al DTO, con gli stessi attributi e getter.
  - *Utilizzo:* Utilizzato nei casi d'uso e nei servizi per manipolare i dati secondo la logica applicativa, mantenendo separazione tra livello di presentazione (DTO) e logica di business (Model).
- **SupportMessageEntity:**
  - *Scopo:* Rappresenta l'entità come viene memorizzata nel database.
  - *Caratteristiche:* Struttura e metodi (getter) simili a DTO e Model.
  - *Utilizzo:* Utilizzata per l'accesso al database, convertendo i dati da modello a entità e viceversa.

**3. Controller**

- **GetSupportMessageController:**
  - *Funzione:* Recupera un singolo messaggio di supporto.
  - *Processo:*
    1. Riceve un SupportMessageDTO.
    2. Converte il DTO in un SupportMessageModel.
    3. Invoca il metodo get_support_message del caso d'uso.
    4. Se il risultato esiste, converte il modello ritornato nuovamente in DTO e lo restituisce.
    5. Gestisce eventuali eccezioni.
- **GetSupportMessagesController:**
  - *Funzione:* Recupera tutti i messaggi di supporto.
  - *Processo:*
    1. Chiama il caso d'uso per ottenere una lista di modelli.
    2. Converte ciascun modello in un SupportMessageDTO.
    3. Restituisce una lista di DTO.
- **SaveSupportMessageController:**
  - *Funzione:* Salva un nuovo messaggio di supporto.
  - *Processo:*
    1. Riceve un DTO in ingresso.
    2. Lo converte in un SupportMessageModel.
    3. Chiama il caso d'uso per salvare il messaggio.
    4. Ritorna l'ID del messaggio salvato.

**4. Use Case e Service**

- **Interfacce (Use Cases):**
  - *GetSupportMessageUseCase, GetSupportMessagesUseCase, SaveSupportMessageUseCase:* Definiscono in modo astratto le operazioni principali (ottenere un messaggio, ottenere tutti i messaggi, salvare un messaggio). Sono classi astratte che impongono la definizione dei metodi necessari.
- **Implementazioni (Service):**
  - *GetSupportMessageService, GetSupportMessagesService, SaveSupportMessageService:* Implementano le interfacce sopra menzionate.
  - Agiscono come "service layer" invocando i metodi definiti nei port (interfacce per l'accesso ai dati) e incapsulando la logica di business.
  - Utilizzano blocchi try/except per gestire eventuali eccezioni e garantire una propagazione controllata degli errori.

**5. Port e Adapter**

- **Port:**
  - *Definizione:* I "port" sono interfacce (ad esempio, GetSupportMessagePort, GetSupportMessagesPort, SaveSupportMessagePort) che definiscono il contratto per l'accesso ai dati.
  - Permettono l'astrazione rispetto al mezzo di persistenza (database, API, ecc.), rendendo l'applicazione indipendente dalla tecnologia usata per il salvataggio o il recupero dei dati.
- **Adapter:**
  - *SupportMessagePostgresAdapter:* Implementa i port sopra citati per ottenere e salvare i messaggi di supporto. Converte tra SupportMessageModel e SupportMessageEntity, garantendo la corretta interazione con il repository.

**6. Repository per PostgreSQL**

- **MessagePostgresRepository:**
  - *Scopo:* Fornisce l'accesso diretto al database PostgreSQL[G] utilizzando la libreria psycopg2.
  - *Metodi principali:*
    - __connect: Metodo privato per stabilire la connessione al database.
    - get_message: Recupera un singolo messaggio basandosi sull'ID.
    - get_messages_by_conversation: Recupera tutti i messaggi relativi a una conversazione.
    - save_message: Inserisce un nuovo messaggio nel database e restituisce l'ID generato.
    - delete_message: Elimina un messaggio dal database.

### 3.4.7 Template

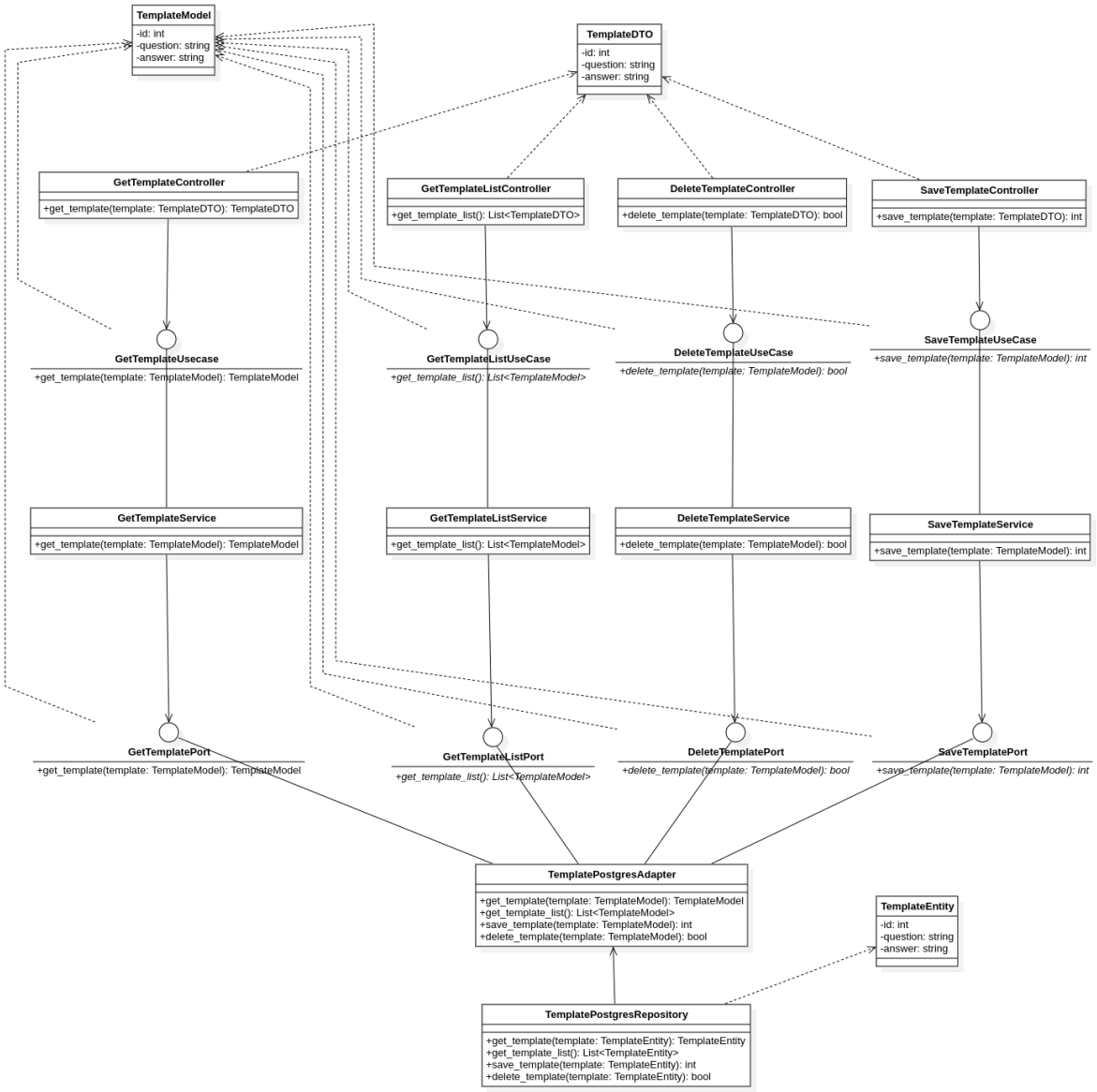


Figura 9: Diagramma delle classi - Template

Il modulo implementa un'architettura a più livelli per la gestione di "template" (che potrebbero rappresentare, ad esempio, dei modelli di domande/risposte) utilizzando i principi della separazione delle responsabilità, dell'iniezione delle dipendenze e dell'astrazione tramite interfacce. Di seguito viene descritto in dettaglio il funzionamento dei vari componenti:

1. **DTO, Model ed Entity**
   - **TemplateDTO:**
     - **Scopo:** È un Data Transfer Object, usato per trasferire dati tra i livelli dell'applicazione (ad esempio, dal controller al livello dei use case e viceversa).
     - **Attributi:** id, question, answer, author_id, last_modified.
     - **Metodi:** I metodi getter (ad esempio, get_id(), get_question(), ecc.) permettono di accedere ai dati incapsulati.

- **TemplateModel:**
  - **Scopo:** Rappresenta il modello di dominio all'interno del business logic. È simile al DTO ma viene usato internamente al sistema per manipolare i dati.
  - **Attributi e metodi:** Gli stessi del DTO, facilitando la conversione e il passaggio dei dati tra i vari layer.
- **TemplateEntity:**
  - **Scopo:** Rappresenta l'oggetto persistente, ovvero il mapping della tabella del database (in questo caso, la tabella "Templates").
  - **Attributi e metodi:** Identici a quelli del DTO e del Model, ma utilizzati specificamente per interagire con il database.

**2. Controllers** I controller fungono da interfaccia tra le richieste dell'utente e i casi d'uso (use cases). Essi ricevono i dati sotto forma di DTO, li convertono in modelli, invocano il caso d'uso corrispondente e poi ritrasformano il risultato in DTO per la risposta.

- **GetTemplateController:**
  - **Funzione:** Riceve un TemplateDTO e crea un TemplateModel corrispondente.
  - **Processo:** Chiama il metodo get_template del use case, controlla se il risultato è None e, in caso contrario, converte il TemplateModel ottenuto in un nuovo TemplateDTO da restituire.
- **GetTemplateListController:**
  - **Funzione:** Recupera l'elenco di tutti i template.
  - **Processo:** Chiama il use case per ottenere la lista di TemplateModel e li trasforma in una lista di TemplateDTO mediante una list comprehension.
- **SaveTemplateController:**
  - **Funzione:** Salva un nuovo template.
  - **Processo:** Converte il TemplateDTO in TemplateModel e invoca il use case per salvare il template. Il metodo restituisce l'ID del template salvato.
- **DeleteTemplateController:**
  - **Funzione:** Gestisce la cancellazione di un template.
  - **Processo:** Converte il TemplateDTO in TemplateModel e chiama il metodo di cancellazione del use case, restituendo un booleano che indica il successo dell'operazione.

**3. Use Cases (Casi d'Uso) e Servizi** I use case definiscono le operazioni fondamentali (ad esempio, ottenere, salvare o cancellare un template) e sono dichiarati come classi astratte per garantire che le implementazioni concrete rispettino la firma definita.

- **Use Case Astratti:**
  - **GetTemplateUseCase, GetTemplateListUseCase, SaveTemplateUseCase, DeleteTemplateUseCase:** Definiscono i metodi astratti che devono essere implementati. Essi forniscono la "contrattualità" per le operazioni che il sistema deve eseguire.

**4. Ports (Interfacce di Ingresso/Uscita)** I "ports" sono interfacce astratte che definiscono come il sistema comunica con l'esterno (ad esempio, il database). Permettono di disaccoppiare la logica di business dall'implementazione concreta della persistenza.

- **GetTemplatePort:** Definisce il metodo get_template per ottenere un template.
- **GetTemplateListPort:** Definisce il metodo get_template_list per recuperare tutti i template.
- **SaveTemplatePort:** Definisce il metodo save_template per salvare un template.
- **DeleteTemplatePort:** Definisce il metodo delete_template per cancellare un template.

**5. Adapter e Repository**

- **TemplatePostgresAdapter:**
  - **Ruolo:** È un adattatore che implementa tutte le interfacce dei ports (Get, List, Save, Delete).
  - **Funzionamento:**
    - Converte un TemplateModel in un TemplateEntity (il formato richiesto dal repository) e viceversa.
    - Invoca i metodi del TemplatePostgresRepository per eseguire le operazioni concrete sul database.
    - Gestisce le conversioni per assicurare che il livello di business lavori con TemplateModel mentre il livello di persistenza lavora con TemplateEntity.
- **TemplatePostgresRepository:**
  - **Scopo:** Fornisce l'accesso diretto al database PostgreSQL[G] utilizzando il modulo psycopg2.
  - **Metodi principali:**
    - __connect(): Stabilisce la connessione al database usando la configurazione fornita.
    - get_template(): Esegue una query per recuperare un template in base al suo ID. Se il template non viene trovato, viene sollevata un'eccezione.
    - get_template_list(): Esegue una query per recuperare tutti i template dalla tabella "Templates" e li restituisce come lista di TemplateEntity.
    - save_template(): Esegue una query di INSERT per salvare un nuovo template. Dopo l'inserimento, restituisce l'ID generato.
    - delete_template(): Esegue una query di DELETE e controlla se almeno una riga è stata eliminata per determinare il successo dell'operazione.

    L'utilizzo dei blocchi with assicura la corretta gestione della connessione e del cursore, mentre il commit della transazione viene eseguito solo se l'operazione di modifica ha avuto successo.

**Riassunto del Flusso dei Dati**

1. **Ricezione della richiesta:** Un client (ad esempio, un'API REST[G]) invoca uno dei controller, passando un TemplateDTO (per operazioni individuali) o richiedendo una lista di template.
2. **Conversione e invocazione del Use Case:** Il controller converte il DTO in un TemplateModel e chiama il relativo use case (implementato nei servizi).
3. **Interazione con il livello di persistenza:** Il servizio, tramite il port, utilizza l'adapter (TemplatePostgresAdapter) per convertire il TemplateModel in TemplateEntity e invoca il metodo corrispondente del repository.

4. **Esecuzione della query sul database:** Il repository stabilisce la connessione, esegue la query SQL, e restituisce il risultato (convertito in TemplateEntity).

5. **Ritorno del risultato:** L'adapter converte l'entity in TemplateModel, il servizio lo passa al controller, che a sua volta lo trasforma in TemplateDTO per inviarlo come risposta al client.

### 3.4.8 Database

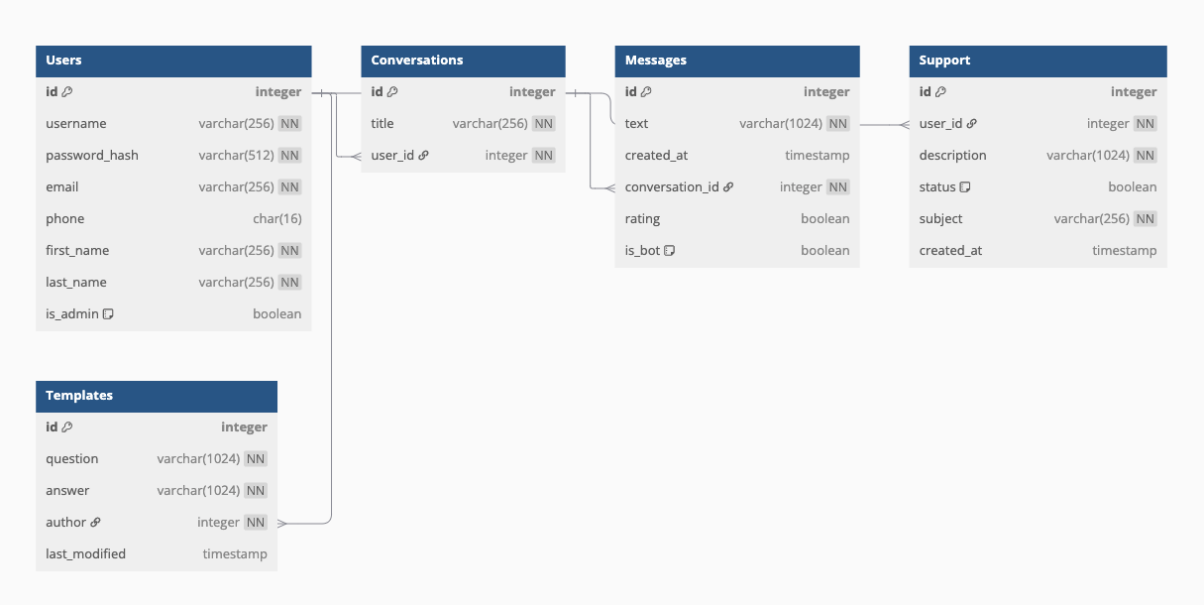


Figura 10: Diagramma ER del Database

Il database è stato progettato per gestire una piattaforma che offre funzionalità di gestione utenti, conversazioni, messaggi, supporto e template. L'obiettivo principale del database è garantire un'archiviazione efficiente e sicura dei dati relativi agli utenti e alle loro interazioni, oltre a supportare funzionalità di supporto tecnico e gestione di risposte preimpostate. La piattaforma è pensata per essere utilizzata da utenti comuni e amministratori, con la possibilità di distinguere tra messaggi generati manualmente e automaticamente (da bot). Ogni conversazione può contenere uno o più messaggi, i quali possono essere valutati tramite un sistema di rating. Inoltre, il database prevede una gestione centralizzata delle richieste di supporto e la possibilità di utilizzare template predefiniti per automatizzare risposte comuni. La struttura del database è stata pensata per garantire integrità e consistenza dei dati attraverso l'uso di vincoli di chiave esterna e l'eliminazione a cascata, in modo da mantenere la pulizia del database ed evitare riferimenti orfani. Di seguito sono descritte nel dettaglio le principali tabelle e la loro struttura.

**Tabella Users**
Questa tabella contiene le informazioni sugli utenti della piattaforma. I campi principali sono:

- **id**: Identificatore univoco dell'utente (generato automaticamente).
- **username**: Nome utente (unico).
- **password_hash**: Hash della password (per la sicurezza).
- **email**: Indirizzo email (unico).
- **phone**: Numero di telefono (opzionale).
- **first_name** e **last_name**: Nome e cognome dell'utente.

- **is_admin**: Flag per indicare se l'utente ha privilegi di amministrazione (valore predefinito: FALSE).

**Tabella Conversations**
Memorizza le conversazioni create dagli utenti. I campi principali sono:

- **id**: Identificatore univoco della conversazione.
- **title**: Titolo della conversazione.
- **user_id**: Riferimento all'utente che ha creato la conversazione.
- Relazione con la tabella **Users**, con eliminazione a cascata.

**Tabella Messages**
Contiene i messaggi all'interno delle conversazioni. I campi principali sono:

- **id**: Identificatore univoco del messaggio.
- **text**: Contenuto del messaggio.
- **created_at**: Data e ora di creazione (timestamp con fuso orario).
- **conversation_id**: Riferimento alla conversazione a cui appartiene.
- **rating**: Valutazione del messaggio (booleano).
- **is_bot**: Flag per indicare se il messaggio è generato da un bot (valore predefinito: FALSE).
- Relazione con la tabella **Conversations**, con eliminazione a cascata.

**Tabella Support**
Gestisce le richieste di supporto inviate dagli utenti. I campi principali sono:

- **id**: Identificatore univoco della richiesta.
- **user_id**: Riferimento all'utente che ha inviato la richiesta.
- **description**: Descrizione del problema o della richiesta.
- **status**: Stato della richiesta (risolta o meno).
- **subject**: Oggetto della richiesta.
- **created_at**: Data e ora della creazione della richiesta.
- Relazione con la tabella **Users**, con eliminazione a cascata.

**Tabella Templates**
Memorizza i template di domande e risposte create dagli utenti (ad esempio, bot o risposte preimpostate). I campi principali sono:

- **id**: Identificatore univoco del template.
- **question** e **answer**: Testo della domanda e della risposta associata.
- **author**: Riferimento all'utente che ha creato il template.
- **last_modified**: Timestamp dell'ultima modifica.
- Relazione con la tabella **Users**, con eliminazione a cascata.

**Eliminazione delle Tabelle**

Sono presenti anche delle query per eliminare tutte le tabelle e i relativi dati in modo sicuro e completo usando il comando `DROP TABLE IF EXISTS ... CASCADE`.

# 4 Tracciamento dei requisiti

In questa sezione vengono descritti i requisiti del sistema e il loro tracciamento. Ogni requisito è identificato da un codice univoco che ne facilita la gestione e il monitoraggio. I requisiti sono suddivisi in categorie in base alla loro natura (funzionali, di qualità, di vincolo) e alla loro importanza (obbligatori, desiderabili, facoltativi). Di seguito viene presentata una tabella che traccia i requisiti funzionali del sistema, indicando per ciascuno di essi il codice identificativo, la descrizione e lo stato di soddisfacimento.
I requisiti sono codificati come segue: **R[Tipo][Importanza][Numero]**
Dove **Tipo** può essere:

- **F (funzionale)**
- **Q (di qualità)**
- **V (di vincolo)**

**Importanza** può essere:

- **O (obbligatorio)**
- **D (desiderabile)**
- **F (facoltativo)**

**Numero** è un numero identificativo univoco del requisito.

## 4.1 Tracciamento requisiti funzionali

| Codice | Descrizione | Stato |
|---|---|---|
| **RFO1** | L'amministratore inserisce dalla pagina di gestione i dati semantici aziendali da cui apprendere la conoscenza da file in formato .pdf. | Soddisfatto |
| **RFO2** | L'amministratore inserisce dalla pagina di gestione i dati semantici aziendali da cui apprendere la conoscenza da file in formato .txt. | Soddisfatto |
| **RFO3** | I testi recuperati dai documenti verranno suddivisi in blocchi, ovvero pezzi più piccoli di dati che rappresentano una piccola porzione del contesto. | Soddisfatto |
| **RFO4** | I vettori generati verranno memorizzati all'interno di un database vettoriale e opportunamente indicizzati. | Soddisfatto |
| **RFO5** | Da un'interfaccia utente della web app, viene catturata una domanda da parte dell'utente. | Soddisfatto |
| **RFO6** | La domanda viene inoltrata al sistema attraverso delle API REST$^{G}$ risiedenti in un Web Server. | Soddisfatto |
| **RFO7** | La rappresentazione vettoriale viene utilizzata per effettuare una ricerca all'interno del database vettoriale da dove vengono reperiti i vettori più simili. | Soddisfatto |
| **RFO8** | La domanda viene inviata al sistema LLM$^{G}$ tramite API. | Soddisfatto |
| **RFO9** | Viene attesa la risposta dall'LLM$^{G}$ tramite API. | Soddisfatto |
| **RFO10** | Attraverso API REST$^{G}$, il sistema inoltra la risposta all'account dell'utente. | Soddisfatto |
| **RFO11** | L'utente deve essere in grado di ottenere informazioni riguardo un prodotto attraverso la conversazione con il bot. | Soddisfatto |
| **RFO12** | L'utente deve essere in grado di ottenere informazioni riguardo una serie di prodotti attraverso la conversazione con il bot. | Soddisfatto |
| **RFO13** | La conversazione tra utente e bot deve essere salvata. | Soddisfatto |
| **RFO14** | L'utente deve essere in grado di visualizzare una delle conversazioni precedentemente salvate. | Soddisfatto |
| **RFO15** | L'utente deve essere in grado di riprendere una delle conversazioni precedentemente salvata. | Soddisfatto |
| **RFO16** | L'utente o l'amministratore devono poter accedere al sistema inserendo Username e Password. | Soddisfatto |
| **RFO17** | L'utente si registra inserendo Username e Password. | Soddisfatto |
| **RFO18** | Gli input del form di registrazione devono essere sanificati per prevenire attacchi SQL Injection. | Soddisfatto |
| **RFO19** | Gli input del form di accesso devono essere sanificati per prevenire attacchi SQL Injection. | Soddisfatto |
| **RFO20** | L'utente deve essere in grado di dare un feedback (thumbsup/thumbsdown) sulla qualità della conversazione dopo averla provata. | Soddisfatto |
| **RFO21** | L'accesso alla dashboard dei "template di domanda e risposta" è consentito solo agli utenti con ruolo di amministratore. | Soddisfatto |
| **RFO22** | Dopo l'accesso da parte dell'amministratore, la pagina di gestione mostra la dashboard dei "template di domanda e risposta". | Soddisfatto |

| Codice | Descrizione | Stato |
|---|---|---|
| **RFO23** | Un "template di domanda e risposta" è formato da una domanda (possibilmente una domanda posta frequentemente che l'amministratore decide di inserire per risparmiare una chiamata al modello) associata ad una corrispondente risposta. | Soddisfatto |
| **RFO24** | L'amministratore deve essere in grado di creare un template, che è formato da una domanda associata ad una corrispondente risposta. | Soddisfatto |
| **RFO25** | L'amministratore deve essere in grado di modificare uno dei template esistenti. | Soddisfatto |
| **RFO26** | L'amministratore deve essere in grado di eliminare un template esistente. | Soddisfatto |
| **RFO27** | Il sistema deve poter fermare la creazione di un template invalido, ovvero quando il template non rispetta il formato Json. | Soddisfatto |
| **RFF28** | L'amministratore deve poter accedere alla dashboard di monitoraggio delle metriche. | Soddisfatto |
| **RFF29** | L'accesso alla dashboard delle metriche delle run è consentito solo agli utenti con ruolo di amministratore. | Non Soddisfatto |
| **RFF30** | Dopo l'accesso da parte dell'amministratore, la pagina di gestione mostra la dashboard delle metriche delle run. | Non Soddisfatto |
| **RFF31** | L'amministratore deve poter selezionare criteri di filtro per visualizzare solo le run di interesse. | Non Soddisfatto |
| **RFF32** | Il sistema deve permettere la selezione di filtri come ID, nome, input, data di inizio e fine, errore, output, tag, numero di token, costo. | Non Soddisfatto |
| **RFF33** | Una volta selezionati i filtri, il sistema deve aggiornare la visualizzazione senza ricaricare l'intera pagina. | Non Soddisfatto |
| **RFF34** | Se nessun filtro è selezionato, il sistema mostra le prime dieci run per impostazione predefinita. | Non Soddisfatto |
| **RFF35** | Dopo aver applicato i filtri, l'amministratore deve poter visualizzare le metriche principali delle run selezionate. | Non Soddisfatto |
| **RFF36** | Il sistema deve mostrare le metriche principali delle run filtrate (ID, nome, input, data di inizio e fine, errore, output, tag, token totali, costo totale). | Non Soddisfatto |
| **RFF37** | La visualizzazione deve essere chiara e strutturata, con possibilità di ordinare le colonne. | Non Soddisfatto |
| **RFO38** | L'amministratore deve poter visualizzare i feedback dati dagli utenti. | Soddisfatto |
| **RFO39** | Il sistema deve poter rifiutare l'importazione dati di file non compatibili, ovvero file non nel formato pdf o txt. | Soddisfatto |
| **RFO40** | L'utente deve poter eliminare una conversazione precedentemente effettuata. | Soddisfatto |
| **RFO41** | L'utente deve poter mandare richieste di assistenza per poter parlare con un operatore umano. | Soddisfatto |
| **RFO42** | L'accesso alla dashboard delle richieste di assistenza è consentito solo agli utenti con ruolo di amministratore. | Soddisfatto |
| **RFO43** | Dopo l'accesso da parte dell'amministratore, la pagina di gestione mostra la dashboard delle richieste di assistenza. | Soddisfatto |

| Codice | Descrizione | Stato |
|---|---|---|
| **RFO44** | L'amministratore deve poter visualizzare le richieste di assistenza ricevute da parte dell'utente. | Soddisfatto |
| **RFO45** | L'amministratore deve poter segnalare ad altri amministratori che una richiesta è stata presa in carico. | Soddisfatto |
| **RFD46** | L'amministratore deve essere in grado di poter rispondere all'utente tramite contatto via e-mail. | Soddisfatto |
| **RFF47** | Le metriche delle run del chatbot devono essere esportabili in JSON. | Non Soddisfatto |
| **RFF48** | Le metriche della run devono includere ID univoco della run, nome assegnato alla sessione, dati di input elaborati dal modello, timestamp di avvio e completamento dell'esecuzione, eventuali errori incontrati, risultato generato dal modello, numero totale di token utilizzati e stima dei costi basata sul consumo di token. | Non Soddisfatto |
| **RFO49** | Il bot per rispondere a una domanda deve ricordarsi i messaggi precedenti nella singola conversazione. | Soddisfatto |
| **RFD50** | Il sistema deve notificare l'utente quando la memoria per le chat salvate è piena e non è possibile salvare ulteriori conversazioni. | Non Soddisfatto |
| **RFO51** | L'utente seleziona una delle domande tra quelle predefinite. | Soddisfatto |
| **RFO52** | L'utente deve essere in grado di visualizzare una lista delle conversazioni precedentemente salvate. | Soddisfatto |
| **RFO53** | La lunghezza massima dell'username è di 256 caratteri. | Soddisfatto |
| **RFO54** | La lunghezza massima della password è di 256 caratteri. | Soddisfatto |
| **RFO55** | Il Sistema rifiuta la registrazione di un nuovo account con username già presente. | Soddisfatto |

Tabella 3: Tabella Requisiti funzionali soddisfatti

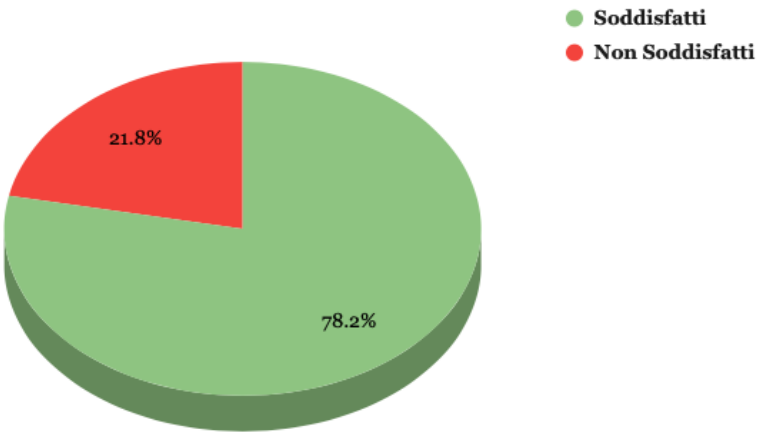


Figura 11: Grafico a torta che mostra la percentuale di soddisfacimento dei requisiti funzionali

## 4.2 Tracciamento requisiti di vincolo

| Codice | Descrizione | Stato |
|---|---|---|
| **RVO1** | Il chatbot deve rispondere con il contesto dato dai file di allenamento (pdf o file di testo inseriti) | Soddisfatto |
| **RVO2** | LLM[G] deve essere integrato tramite API | Soddisfatto |
| **RVO3** | LLM[G] utilizzato deve essere quello di OpenAI | Soddisfatto |
| **RVO4** | Deve essere usato un database relazionale | Soddisfatto |
| **RVO5** | Deve essere gestito il salvataggio delle chat precedenti con tutti i messaggi in esse tramite un database relazionale con PostgreSQL[G] | Soddisfatto |
| **RVO6** | Deve essere implementato un database vettoriale | Soddisfatto |
| **RVO7** | Deve essere implementato un database vettoriale FAISS[G] per poter rendere possibile la ricerca con contesto dall'LLM[G] | Soddisfatto |
| **RVO8** | Deve essere implementato un embedding model | Soddisfatto |
| **RVO9** | L'embedding model deve essere quello di OpenAI | Soddisfatto |
| **RVO10** | Deve essere implementata una WebApp che permetta di comunicare con il chatbot | Soddisfatto |
| **RVO11** | L'interfaccia deve essere costruita utilizzando componenti funzionali React | Soddisfatto |
| **RVO12** | Si deve creare un backend che gestisca le chiamate HTTP, il database vettoriale e il database relazionale con Flask | Soddisfatto |
| **RVO13** | La gestione dello stato locale deve essere implementata tramite useState | Soddisfatto |
| **RVO14** | La WebApp deve utilizzare React Router per gestire la navigazione tra le pagine | Soddisfatto |
| **RVO15** | Gli stili devono essere gestiti tramite CSS inline o con className per garantire modularità | Soddisfatto |
| **RVO16** | La comunicazione tra componenti deve essere gestita inviando funzioni come props | Soddisfatto |
| **RVO17** | La WebApp deve essere responsiva e adattarsi dinamicamente alle dimensioni della finestra | Soddisfatto |
| **RVO18** | La gestione dei blocchi di testo vettorializzati deve essere gestita tramite Faiss | Soddisfatto |
| **RVD19** | Le metriche delle run del chatbot devono essere recuperate tramite Langsmith | Soddisfatto |
| **RVO20** | Bisogna usare la libreria LangChain[G] per la interazione con i modelli LLM[G] e Embedding[G] | Soddisfatto |

Tabella 4: Tabella Requisiti di vincolo soddisfatti

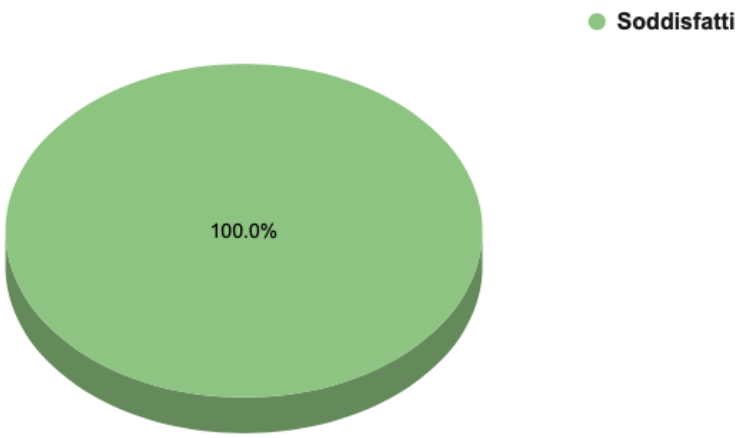


Figura 12: Grafico a torta che mostra la percentuale di soddisfacimento dei requisiti di vincolo

## 4.3 Tracciamento requisiti di qualità

| Codice | Descrizione | Stato |
|---|---|---|
| **RQO1** | Schema di progettazione della base di dati | Soddisfatto |
| **RQO2** | Codice prodotto in formato sorgente reso disponibile tramite repository pubblici | Soddisfatto |
| **RQO3** | Documentazione riassuntiva delle metriche e dei risultati | Soddisfatto |
| **RQO4** | Il software deve essere testato con una copertura di codice minima dell'80% e una copertura dei rami dell'80%, con un obiettivo ottimale del 100% | Soddisfatto |
| **RQO5** | Il 90% dei test deve essere superato come requisito minimo, mentre l'obiettivo ottimale è il 100% | Soddisfatto |
| **RQO6** | La metodologia di sviluppo deve seguire il paradigma del Test Driven Development (TDD), garantendo che il codice venga scritto partendo dai test | Soddisfatto |

Tabella 5: Tabella Requisiti di qualità soddisfatti

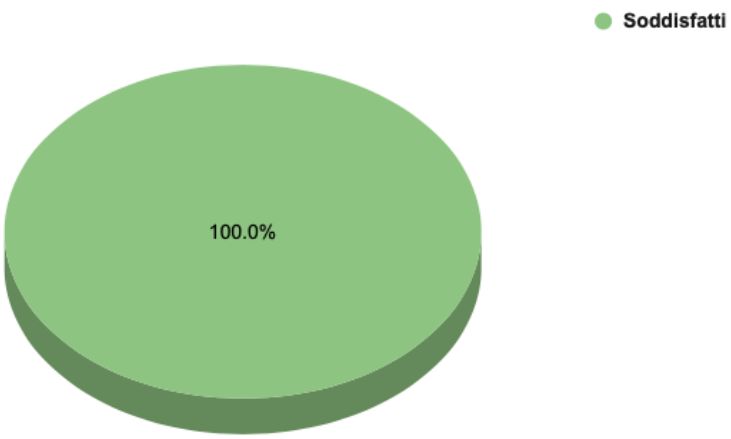


Figura 13: Grafico a torta che mostra la percentuale di soddisfacimento dei requisiti di qualità

## 4.4 Soddisfazione totale dei requisiti

Il gruppo Code7Crusaders ha soddisfatto 69 su 81, arrivando ad una copertura del **85.2%**.

| Soddisfatti | Non soddisfatti |
|---|---|
| 69 | 12 |

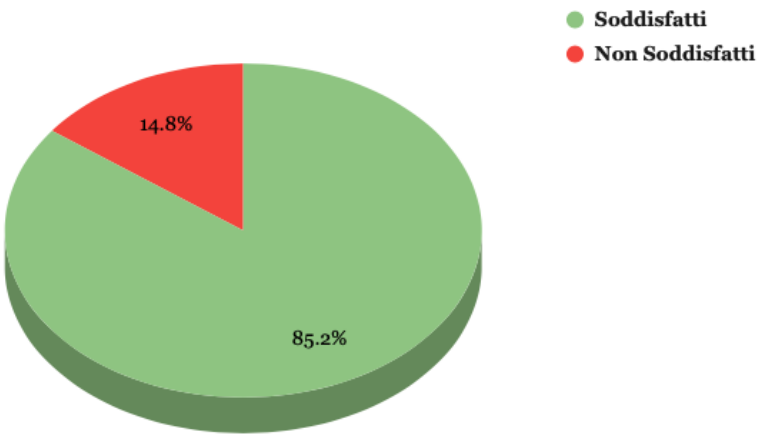


Figura 14: Grafico a torta dei Requisiti soddisfatti rispetto al totale

Per quanto riguarda la copertura dei requisiti obbligatori, la copertura rilevata è di 66 su 66 requisiti, arrivando quindi ad un **100%** sul totale.

| Soddisfatti | Non soddisfatti |
|---|---|
| 66 | 0 |

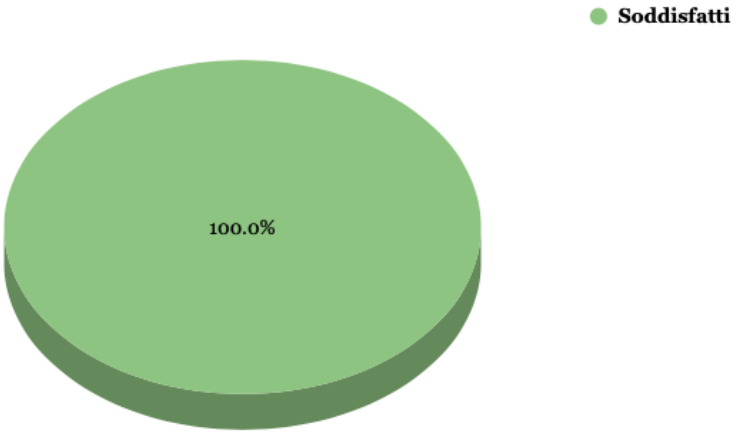


Figura 15: Grafico a torta dei Requisiti obbligatori soddisfatti rispetto al totale

In termini di soddisfacimento dei requisiti desiderabili, è stata raggiunta una copertura del **66.67%**, con 2 su 3.

| Soddisfatti | Non soddisfatti |
|---|---|
| 2 | 1 |

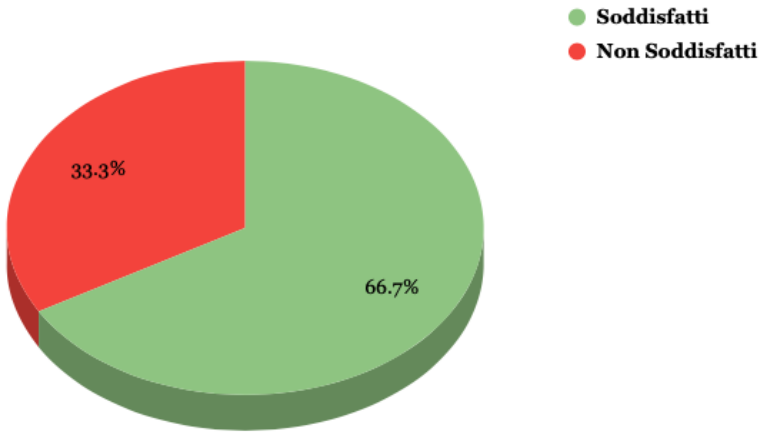


Figura 16: Grafico a torta dei Requisiti desiderabili soddisfatti rispetto al totale

Per quanto concerne l'adempimento dei requisiti facoltativi, abbiamo conseguito una percentuale del **8.3%** sul totale, con 1 su 12 requisiti considerati.

| Soddisfatti | Non soddisfatti |
|---|---|
| 1 | 11 |

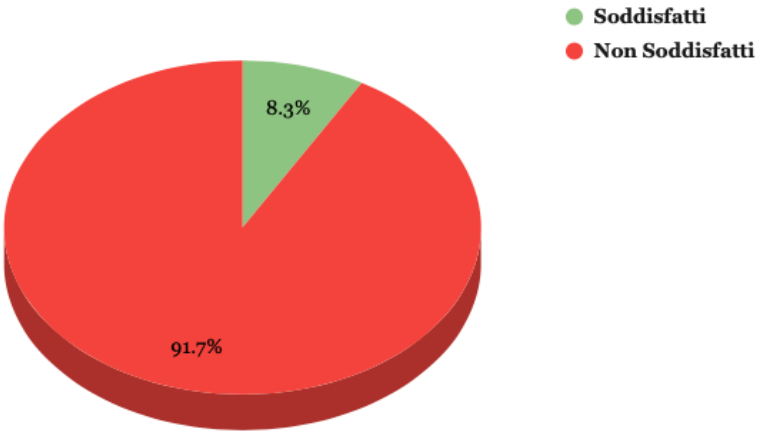


Figura 17: Grafico a torta dei Requisiti facoltativi soddisfatti rispetto al totale