Specifica Tecnica

2025-04-15

V1.0.0

sweetenteam@gmail.com

https://sweetenteam.github.io

Destinatari	Prof. Tullio Vardanega Prof. Riccardo Cardin AzzurroDigitale
Redattori	Valeri Mihail Belenkov Davide Benedetti Orlando Ferazzani Nicolas Fracaro Mouad Mahdi Andrea Santi
Verificatori	Valeri Mihail Belenkov Davide Benedetti Matteo Campagnaro Nicolas Fracaro Mouad Mahdi Andrea Santi

Registro delle modifiche

Versione	Data	Autori	Verificatori	Dettaglio
1.0.0	2025-04-15	Valeri Mihail Belenkov	Mouad Mahdi	Approvazione per PB
0.0.9	2025-04-15	Valeri Mihail Belenkov	Nicolas Fracaro	Aggiunta documentazione raccolta e salvataggio informazioni da Github e Confluence
0.0.8	2025-04-12	Nicolas Fracaro	Valeri Mihail Belenkov	Aggiunta sezioni: Intorduzione microservizio Informazioni, Recupero informazioni dei dati da Jira e Recupero delle informazioni rilevanti basato su query dell'utente
0.0.7	2025-04-10	Davide Benedetti	Mouad Mahdi	Stesura tracciamento stato requisiti funzionali
0.0.6	2025-04-09	Andrea Santi	Mouad Mahdi	Stesura microservizio denominato «Storico»
0.0.5	2025-04-08	Davide Benedetti	Nicolas Fracaro	Aggiunta sezione microservizio ChatBot
0.0.4	2025-04-03	Nicolas Fracaro	Andrea Santi	Sezione architettura di sistema e introduzione backend
0.0.3	2025-03-27	Orlando Ferazzani	Matteo Campagnaro	Aggiunte tecnologie di testing e miglioramenti generali
0.0.2	2025-03-15	Orlando Ferazzani	Matteo Campagnaro	Aggiunta sezione «Architettura frontend»
0.0.1	2025-02-27	Mouad Mahdi	Davide Benedetti	Stesura sezione microservizio Api-Gateway

Indice

Lista della immagini

1) Introduzione

1.1) Scopo del documento

Il presente documento ha lo scopo di fungere da risorsa esaustiva per la spiegazione e conseguente comprensione degli aspetti tecnici del progetto azzurrodigitale:

Figura 1: Logo BuddyBot

La sua finalità primaria è quella di fornire una panoramica dettagliata e approfondita delle scelte progettuali, architetturali e tecnologiche del sistema sviluppato. In particolare, si intende fornire un'analisi profonda estesa al livello di progettazione più basso, includendo spiegazione, definizione e motivazione delle scelte effettuate, e dei *design pattern*$_G$ adottati.

Il documento ha quindi scopi molteplici:

- Motivare le scelte progettuali e di sviluppo adottate;
- Fungere da guida per il processo di sviluppo e manutenzione del sistema;
- Fornire una vista panoramica e monitorare la *Code Coverage*$_G$ dei requisiti del progetto identificati nel documento `Analisi dei Requisiti` (visionabile qui);

L'adeguatezza e la completezza del documento (e del progetto) sono in costante evoluzione e miglioramento in base ai *feedback*$_G$ ricevuti e sulla base dell'aggiornamento dei requisiti.

1.2) Scopo del prodotto

L'obiettivo del progetto è la realizzazione di un *chatbot*$_G$ sotto forma di *Web App*$_G$ atto a fornire un supporto al team di azzurrodigitale: nella gestione delle attività di un progetto in corso di sviluppo. Nella fattispecie, il chatbot utilizza delle *API*$_G$ e un modello di *LLM*$_G$ per, rispettivamente, reperire informazioni da sistemi esterni utilizzati dall'azienda (più specificatamente, Jira, GitHub e Confluence) e elaborare una risposta. Questa risposta può contenere del semplice testo, un link o un *code block*$_G$. Il chatbot ha una singola sessione per ogni utente, e può essere utilizzato da più utenti contemporaneamente.

Il team è confidente che questo genere di prodotto migliorerà il workflow del team di azzurrodigitale:, riducendo i tempi di risposta e migliorando la qualità del lavoro svolto.

1.3) Miglioramenti e maturità

Questo documento è redatto con approccio incrementale e modificato nel tempo per riflettere l'andamento del progetto e le decisioni prese. In particolare, il documento è soggetto a modifiche in base ai feedback ricevuti e all'evoluzione dei requisiti del progetto. Per questo motivo, il documento non è considerabile definitivo, esaustivo e completo fino al raggiungimento di una versione stabile dello stesso (`1.0.0` o superiore).

1.4) Glossario

Per evitare ambiguità e incomprensione riguardanti la terminologia tecnica utilizzata nel documento, viene redatto e adottato un Glossario contenente le definizioni dei termini tecnici utilizzati. Il Glossario è consultabile qui e i termini presenti nel documento sono evidenziati con *questo stile*$_G$.

1.5) Riferimenti

1.5.1) Riferimenti normativi

- Presentazione pdf del capitolato C9: C9p.pdf (*versione disponibile al 2025-03-20*)
- Norme di Progetto: Norme_di_Progetto_v1.0.0.pdf
- Piano di Qualifica: Piano_di_Qualifica_v1.0.0.pdf

1.5.2) Riferimenti informativi

- Analisi dei Requisiti: Analisi_dei_Requisiti_v1.1.0.pdf
- Glossario: Glossario
- I diagrammi dei casi d'uso: Use case
- Progettazione: I pattern architetturali Software Architecture Patterns
- Verifica e validazione: analisi statica (T10): analisi statica
- Verifica e validazione: analisi dinamica aka testing (T11): analisi dinamica
- Programmazione: SOLID programming principles

1.5.3) Riferimenti Tecnici

- Documentazione ufficiale Typescript: Typescript
- Documentazione ufficiale Langchain: Langchain
- Documentazione ufficiale NodeJs: Node.js
- Documentazione ufficiale NestJs: Nest.js
- Documentazione ufficiale Groq: GroqCloud
- Documentazione ufficiale Qdrant: Qdrant
- Documentazione ufficiale NomicAi: NomicAi
- Documentazione ufficiale PostgreSQL: PostgresSQL
- Documentazione ufficiale Oktokit: Octokit
- Documentazione JiraJs: JiraJs
- Documentazione Confluence Js: ConfluenceJs
- Documentazione ufficiale Docker: Docker
- Documentazione ufficiale ReactJs: React
- Documentazione ufficiale ReactQuery (TanStack) ReactQuery
- Documentazione ufficiale TailwindCSS: Tailwind CSS
- Documentazione ufficiale NextJs Next.js

2) Tecnologie

In questo capitolo sono elencate tutte le tecnologie della ***tech stack***$_G$ che il team utilizza per lo sviluppo del progetto di azzurrodigitale;, come linguaggi di programmazione, ***framework***$_G$, ***librerie***$_G$ e ***ambienti di sviluppo***$_G$.

2.1) Tecnologie di sviluppo

2.1.1) Typescript

Typescript è un linguaggio di programmazione open-source. È un super-set di JavaScript, che aggiunge forte tipizzazione statica. Il team ha scelto di utilizzare Typescript per la sua tipizzazione statica, che permette di ridurre gli errori di programmazione e di rendere il codice più leggibile e manutenibile.

Figura 2: Logo Typescript

2.1.2) Langchain

Langchain è un framework open-source per la creazione di applicazioni basate sull'utilizzo *LLM*$_G$. Il team ha scelto di utilizzare Langchain per la sua facilità d'uso e per la sua integrazione con altri servizi come Qdrant e Groq, oltre che ad avere una libreria in Typescript, rendendolo compatibile con il nostro linguaggio.

Figura 3: Logo di Langchain

2.1.3) Node.js

Node.js è un ambiente di runtime open-source per l'esecuzione di codice JavaScript lato server. Il team ha scelto di utilizzare Node.js per la sua scalabilità e per la sua facilità di utilizzo.

Figura 4: Logo di Node.js

2.1.4) Nest.js

Nest.js è un framework per la creazione di applicazioni server-side in Node.js. Il team ha scelto di utilizzare Nest.js per la sua struttura modulare e per la sua scalabilità e per la facilità con cui è possibile creare i design pattern più opportuni.

Figura 5: Logo di Nest.js

2.1.5) GroqCloud

È una piattaforma AI basata su hardware specializzato (LPU) per inferenza ad alte prestazioni, supporta modelli LLM e integrazione con strumenti AI per elaborazione in tempo reale.

Figura 6: Logo di GroqCloud

2.1.6) Qdrant

Qdrant è un motore di ricerca e analisi di dati non strutturati, supporta l'indicizzazione e la ricerca di dati in tempo reale, oltre che la ricerca di dati basata su vettori.

Figura 7: Logo di Qdrant

2.1.7) NomicAi

NomicAi è un servizio di elaborazione del linguaggio naturale (NLP) basato su modelli LLM che permette l'embedding di testo. Il team ha scelto di utilizzare NomicAi per la sua facilità d'uso e per la sua integrazione con altri servizi come Langchain e Groq.

Figura 8: Logo di NomicAi

2.1.8) PostgreSQL

PostgreSQL è un sistema di gestione di database relazionale open-source. Il team ha scelto di utilizzare PostgreSQL per la sua affidabilità e per la sua estensiva documentazione.

Figura 9: Logo di PostgreSQL

2.1.9) Octokit

Octokit è un toolkit per l'interazione con le API di GitHub. Il team ha scelto di utilizzare Octokit per la sua estesa documentazione e per utilizzare un prodott ufficiale per interagire on GitHub stesso.

Figura 10: Logo di Octokit

2.1.10) JiraJs

JiraJs è un toolkit per l'interazione con le API di Jira. Il team ha scelto di utilizzare JiraJs per la sua documentazione affidabile e per la sua facilità d'uso.

Figura 11: Logo di JiraJs

2.1.11) ConfluenceJs

ConfluenceJs è un toolkit per l'interazione con le API di Confluence. Il team ha scelto di utilizzare ConfluenceJs per la sua documentazione affidabile e per la sua facilità d'uso.

Figura 12: Logo di ConfluenceJs

2.1.12) Docker

Docker è una piattaforma open-source per lo sviluppo, il deploy e l'esecuzione di applicazioni in container. Il team ha scelto di utilizzare Docker per la sua facilità di deploy e per la sua scalabilità.

Figura 13: Logo di Docker

2.1.13) React.js

ReactJs è una libreria open-source per la creazione di interfacce utente. Il team ha scelto di utilizzare ReactJs per la sua immediatezza nell'uso, per la sua scalabilità e per la sua estesa documentazione.

Figura 14: Logo di ReactJs

2.1.14) ReactQuery

ReactQuery è una libreria open-source per la gestione dello stato in React. Il team ha scelto di utilizzare ReactQuery per la sua integrazione con React.

Figura 15: Logo di ReactQuery

2.1.15) TailwindCSS

TailwindCSS è un framework CSS utilizzato per la creazione di interfacce utente. Il team ha scelto di utilizzare TailwindCSS per la sua facilità d'uso e per la sua documentazione dettagliata oltre che per utilizzare una tecnologia più compatibile con il resto.

Figura 16: Logo di TailwindCSS

2.1.16) Next.js

Next.js è un framework per la creazione di applicazioni web in React. Il team ha scelto di utilizzare Next.js per i metodi nativi a disposizione per le richieste alle API e per utilizzare una tecnologia più nuova rispetto al resto.

Figura 17: Logo di Next.js

2.1.17) ShadCn

Libreria di componenti pre-impostati, pronti all'uso e altamente customizzabili. Il team ha scelto di utilizzare ShadCn per la sua facilità d'uso e per la sua documentazione dettagliata, oltre che per sfruttare al massimo il principio del riuso.

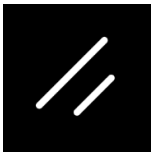

Figura 18: Logo di ShadCn

2.1.18) LucideReact

Libreria di icone SVG pronte all'uso. Il team ha scelto di utilizzare LucideReact per la sua facilità d'uso e per la sua documentazione dettagliata, oltre che per sfruttare al massimo il principio del riuso.

Figura 19: Logo di LucideReact

2.2) Tecnologie di testing

2.2.1) Jest

Jest è un framework di testing per JavaScript. Il team ha scelto di utilizzare Jest per la sua facilità d'uso e per la sua integrazione con Typescript. Utilizzato per **Analisi dinamica** in quanto richiede l'esecuzione del codice.

Figura 20: Logo di Jest

2.2.2) ESLint

ESLint è uno strumento di analisi statica del codice per identificare e segnalare errori di programmazione. Il team ha scelto di utilizzare ESLint per la sua facilità d'uso e per la sua integrazione con Typescript. Utilizzato per **Analisi statica** in quanto non richiede l'esecuzione del codice.

Figura 21: Logo di ESLint

3) Architettura di Sistema

3.1) Approccio alla Progettazione

La progettazione dell'architettura di sistema di *BuddyBot* è stata condotta secondo un approccio *top-down*. Questo metodo ha permesso di definire inizialmente i macro-componenti del sistema, garantendo una visione chiara e coerente sin dalle prime fasi. Successivamente, si è passati a un raffinamento progressivo delle specifiche dei singoli moduli e componenti, assicurando che ciascuno fosse progettato in modo modulare e scalabile. Tale approccio ha facilitato la suddivisione delle responsabilità tra i membri del team, migliorando la tracciabilità delle decisioni progettuali.

3.2) Contenitori e Deploy con Docker

Per garantire portabilità e facilitare il deploy, è stato adottato Docker e Docker Compose, con un container per ogni servizio e per le risorse di supporto.

L'utilizzo di Docker porta molti vantaggi, tra cui:

- **Isolamento dei servizi**: Ogni microservizio gira in un ambiente indipendente, evitando conflitti tra dipendenze.
- **Portabilità**: Il sistema può essere eseguito su qualsiasi piattaforma senza configurazioni complesse.
- **Facilità di scalabilità**: Può essere facilmente distribuito su più istanze per gestire carichi elevati.
- **Coerenza ambientale**: Assicura che gli ambienti di sviluppo, test e produzione siano identici, riducendo i problemi legati a differenze di configurazione.

Docker Compose viene utilizzato per orchestrare e avviare automaticamente i container, garantendo l'interconnessione tra i microservizi e i database senza necessità di configurazioni manuali complesse.

4) Architettura di sistema

4.1) Strutturazione Generale del Sistema

Il sistema è stato suddiviso in due macro-componenti principali:

- **Frontend**: Interfaccia utente per l'interazione con BuddyBot.
- **Backend**: Gestione della logica applicativa e delle fonti dati, esposto tramite API REST.

Questa suddivisione consente di ottenere diversi benefici:

- Indipendenza tra frontend e backend: Gli aggiornamenti possono avvenire separatamente, evitando impatti sull'intero sistema.
- Possibilità di supportare frontend multipli: L'uso di *API REST* consente l'integrazione di differenti interfacce utente, come web app, mobile app e desktop app (anche se attualmente non implementato, questa architettura lo renderebbe facilmente realizzabile in futuro).
- Scalabilità e manutenibilità migliorate: Il backend può evolvere indipendentemente dall'interfaccia utente, permettendo di migliorare le prestazioni senza dover aggiornare ogni client.

4.2) Architettura del frontend

Per la parte di frontend, il team ha utilizzato ***Next.Js***$_G$, framework basato su React, per la creazione di pagine web. Next.Js è stato scelto per la sua facilità d'uso e per la sua scalabilità. Inoltre, il team ha utilizzato ***TailwindCSS***$_G$ per la creazione di interfacce utente. TailwindCSS è stato scelto per la sua

facilità d'uso e per la sua documentazione dettagliata, oltre che per la semplificazione della specificità di CSS base.

La scelta di tali tecnologie ha portato il team ad uno sviluppo a componenti del frontend. Saranno questi poi a comporre la struttura della web app. L'approccio a componenti, tipico di React, permette una maggiore modularità e scalabilità del codice, oltre che ad una maggiore facilità di manutenzione, evitando di avere tutto il codice in una singola pagina.

BuddyBot è una ***SPA***$_G$, ovvero una Single Page Application, che permette di avere una sola pagina web che viene caricata una sola volta e che viene aggiornata dinamicamente senza dover ricaricare la pagina. Questo permette di avere una maggiore velocità di caricamento e di navigazione all'interno della web app. Inoltre, essendo un ChatBot, non vi era la necessità di avere più di una pagina, anche se il team ha previsto la possibilità di aggiungere nuove pagine in futuro.

4.3) Architettura del Backend

4.3.1) Architettura di Deployment

Il backend è strutturato secondo un'architettura a *microservizi*, dove ogni servizio è responsabile di una specifica funzionalità del sistema. Questo approccio ha permesso di ottenere un sistema più modulare e scalabile, pur affrontando alcune sfide specifiche.

4.3.1.1) Vantaggi dell'architettura a microservizi

- Scalabilità orizzontale: I microservizi possono essere replicati per gestire carichi di lavoro elevati.
- Indipendenza di deploy: Ogni servizio può essere aggiornato, riavviato o sostituito senza impattare il resto del sistema.
- Manutenibilità e modularità: Separare le funzionalità in microservizi facilita la gestione del codice e l'aggiunta di nuove feature.
- Tecnologie eterogenee: Ogni microservizio può essere sviluppato con la tecnologia più adatta senza vincoli imposti da un monolite.

4.3.1.2) Svantaggi

- Overhead di gestione: A differenza di un'architettura monolitica, i microservizi richiedono una gestione più complessa, sia in fase di sviluppo che di deploy.
- Comunicazione tra servizi: Per garantire un'integrazione efficiente, è stato necessario implementare un sistema di messaggistica asincrono, come *RabbitMQ*, per la comunicazione tra microservizi.

4.3.1.3) Microservizi Identificati

Il backend è suddiviso in quattro microservizi principali:

- **API Gateway**: Instrada le richieste tra frontend e microservizi interni, gestisce il bilanciamento del carico e pianifica il recupero delle informazioni dalle fonti.
- **Chatbot**: Genera risposte basandosi sulle richieste ricevute e sulle informazioni contestuali fornite dal database vettoriale.
- **Storico**: Salva e recupera le domande e le risposte dal database relazionale (*PostgreSQL*) per mantenere uno storico delle conversazioni.
- **Information Vector DB**: Recupera informazioni dalle fonti, effettua embedding in forma vettoriale e le memorizza nel database vettoriale (*Qdrant*), fornendo dati contestuali al chatbot.

4.3.1.4) Comunicazione tra Microservizi: RabbitMQ

Nell'architettura a microservizi di ***BuddyBot***$_G$, la comunicazione efficiente tra componenti è garantita da un sistema di messaggistica asincrona basato su RabbitMQ.

L'adozione di RabbitMQ offre benefici fondamentali:

- **Flessibilità temporale:** Determina quando un microservizio elabora una richiesta, eliminando blocchi nell'esecuzione.
- **Scalabilità orizzontale:** I messaggi vengono distribuiti in code ed elaborati in parallelo.
- **Resilienza avanzata:** I messaggi persistono nelle code quando i servizi destinatari sono temporaneamente non disponibili.
- **Disaccoppiamento:** Riduce le dipendenze dirette tra microservizi, semplificando la manutenzione.

4.3.1.4.1) Pattern e implementazione

Il sistema utilizza principalmente il pattern **RPC asincrono (Request/Response)** per le comunicazioni tra i microservizi, sfruttando l'integrazione tra NestJS e RabbitMQ:

- NestJS gestisce automaticamente gli identificativi di correlazione tra richieste e risposte.
- Il framework `@nestjs/microservices` fornisce astrazioni per configurare microservizi basati su code.
- Ogni microservizio implementa:
  - Listener dedicati che si connettono a specifiche code RabbitMQ.
  - Handler che associano pattern predefiniti alle funzioni di business logic.
  - Client per pubblicare messaggi in modo asincrono.

4.3.2) Architettura logica

Il sistema è progettato seguendo l'**architettura esagonale**, un modello architetturale che crea una separazione netta tra la business logic dell'applicazione e il mondo esterno, garantendo indipendenza da tecnologie specifiche e maggiore manutenibilità.

4.3.2.1) Struttura dell'architettura esagonale

Logica di business rappresenta il nucleo dell'applicazione, contenente il dominio e le regole di business. È completamente indipendente da implementazioni tecnologiche specifiche, garantendo massima portabilità e riutilizzabilità.

Porte definiscono i punti di interazione tra il nucleo e il mondo esterno:
- **Porte in Entrata (Use Case)**: Permettono ai componenti esterni di invocare il nucleo, fornendo un accesso strutturato e proteggendo la logica di dominio da implementazioni specifiche.
- **Porte in Uscita**: Consentono al nucleo di accedere a funzionalità esterne (database, servizi di terze parti) mantenendo l'astrazione tecnologica.

Services implementano le porte in entrata e fanno parte della business logic. Si concentrano esclusivamente sulla logica di dominio, rimanendo indipendenti da aspetti tecnologici specifici.

Adapters costituiscono il livello più esterno dell'applicazione e si dividono in:
- **Adapters in Entrata (Controller)**: Gestiscono e convertono le richieste provenienti dall'esterno verso il core.
- **Adapters in Uscita**: Gestiscono la comunicazione dal core verso servizi e tecnologie esterne.

4.3.2.2) Vantaggi

Questa architettura garantisce:

- **Flessibilità**: L'applicazione rimane indipendente dalle tecnologie esterne, facilitando modifiche e aggiornamenti senza impattare la logica di business.
- **Testabilità**: La logica di business può essere testata in isolamento, semplificando lo sviluppo test-driven.
- **Resilienza**: Il sistema diventa più resistente ai cambiamenti tecnologici, permettendo di sostituire componenti esterni senza modificare il nucleo applicativo.

4.3.3) Design pattern utilizzati

4.3.3.1) Dependency Injection

Uno degli aspetti fondamentali dell'implementazione del backend è stato l'uso del pattern di **Dependency Injection**, nativamente supportato da NestJS. Questo approccio ha permesso di ridurre l'accoppiamento tra i componenti, semplificando la testabilità e la manutenzione del sistema spostando all'esterno della classi la risoluzione delle dipendenze.

NestJS adotta un **container per le dipendenze** che consente di dichiarare i provider una sola volta e iniettarli ovunque siano richiesti tramite il costruttore delle classi. Ogni modulo dell'applicazione può registrare provider, che vengono poi risolti automaticamente dal framework quando una classe dichiara di averne bisogno.

5) Progettazione di dettaglio

5.1) Progettazione frontend

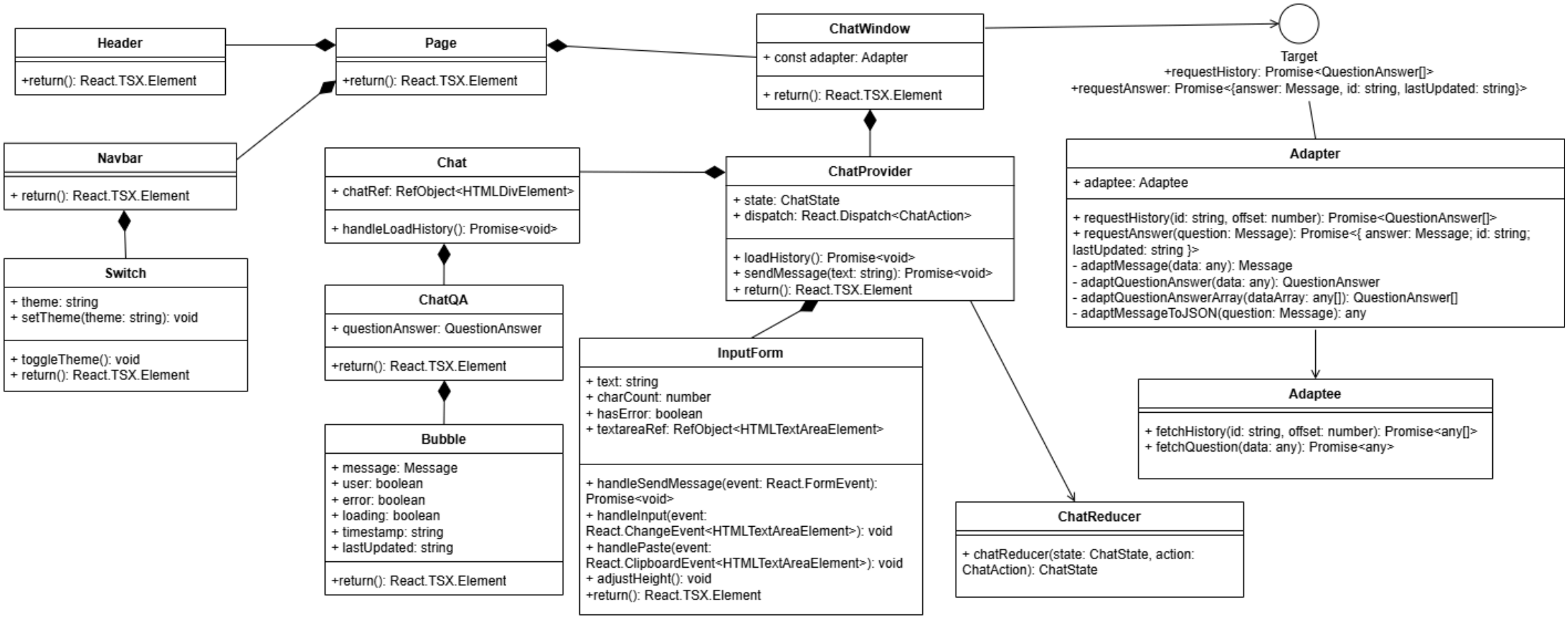

Figura 22: UML frontend

5.2) Architettura nel dettaglio

5.2.1) Componenti

Come detta lo standard di `Next.JS`, la pagina principale è `page.tsx`, che contiene la struttura base della web app. All'interno di questa pagina, vengono poi importati i vari componenti che compongono la web app. I componenti principali sono:

- **`Header.tsx`** è progettato per mostrare il logo e il nome dell'applicazione in modo ben visibile in cima alla pagina, contribuendo immediatamente a definire l'identità visiva della web app.

Figura 23: Header della pagina in dark mode

- **`Navbar.tsx`** gestisce la navigazione; anche se BuddyBot è una Single Page Application, la navbar offre all'utente la possibilità di accedere rapidamente ad altri siti web utili.

Figura 24: Navbar della pagina in dark mode

- **`ChatWindow.tsx`** integra due componenti distinti: il componente **`Chat.tsx`** per visualizzare lo storico della conversazione e l'**`InputForm.tsx`** per l'inserimento dei messaggi, creando così un'unica area interattiva per gestire la chat.

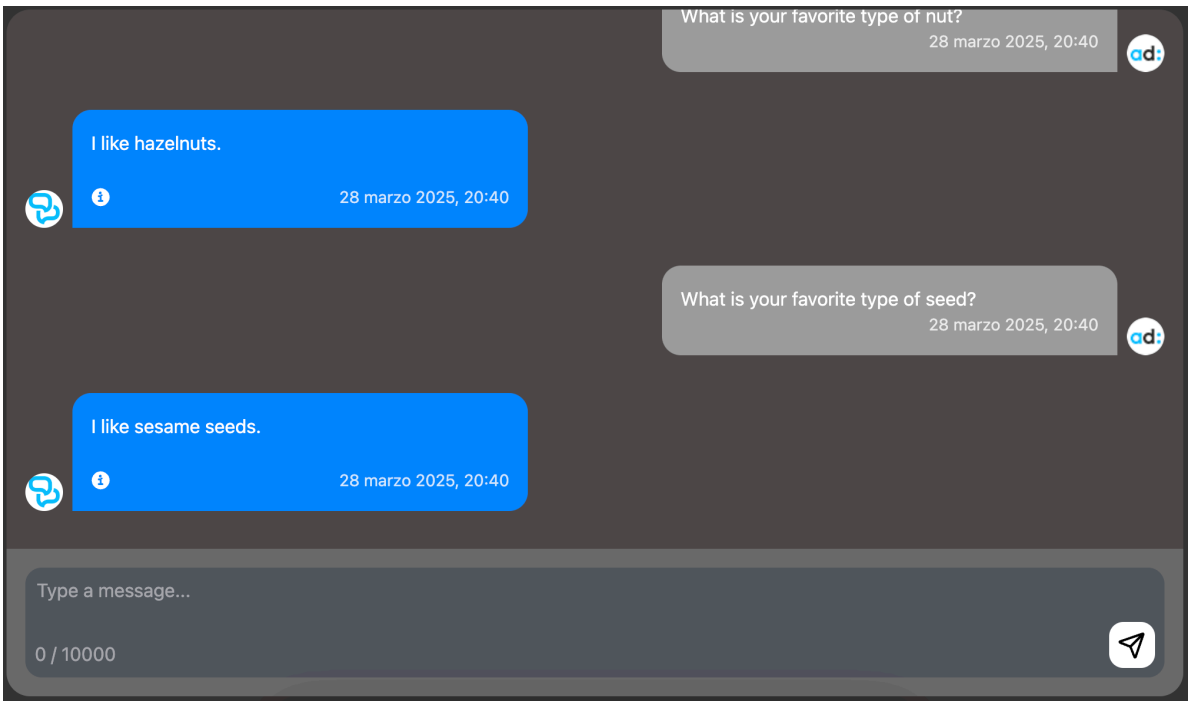

Figura 25: ChatWindow della pagina in dark modse

- **`Chat.tsx`** si occupa di mostrare l'intera conversazione tra utente e chatbot. Al suo interno, ogni scambio è rappresentato da un componente **`ChatQA.tsx`**, che racchiude due **`Bubble.tsx`**: una per il messaggio dell'utente e una per la risposta generata dal chatbot.

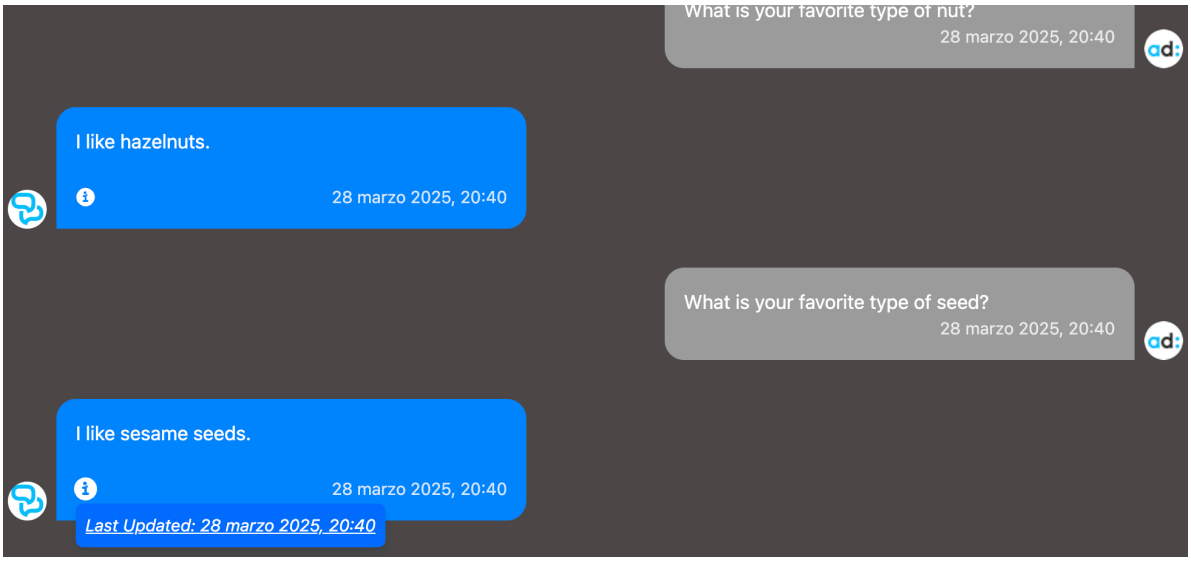

Figura 26: Chat della pagina in dark mode

Ci sono inoltre altre componenti, utilizzate a supporto dei componenti principali. Questi sono inclusi nella cartella denimoinata `ui`. Queste componenti sono:

- `Button.tsx`;
- `LoadMessage.tsx`;
- `LoadChat.tsx`;
- `ErrorAlert.tsx`;
- `InfoAlert.tsx`;
- `MarkDown.tsx`;
- `Avatar.tsx`;
- `MessageAvatar.tsx`;

5.2.2) Struttura dei dati

Per sviluppare al meglio e più dettagliatamente il team ha definito dei tipi, che gestiscono diversi aspetti della web app. Questi tipi sono definiti all'interno della cartella `types` e sono:

- **`Action.ts`**
  Definisce le possibili azioni che la chat può eseguire (es. caricamento della cronologia, aggiunta di messaggi, gestione degli errori).

```ts
import { Message } from "@/types/Message";
import { QuestionAnswer } from "./QuestionAnswer";

export type ChatAction =
  | { type: "LOAD_HISTORY_START" }
  | { type: "LOAD_HISTORY_SUCCESS"; payload: QuestionAnswer[], hasMore: boolean }
  | { type: "LOAD_HISTORY_ERROR", error: number }
  | { type: "ADD_MESSAGE_START"; id: string, question: Message }
  | { type: "ADD_MESSAGE_SUCCESS"; id: string, answer: Message, newid: string, lastUpdated: string }
  | { type: "ADD_MESSAGE_ERROR"; id: string, error: number }
  | { type: "SCROLL_DOWN" };
```

- **`ChatContext.ts`**
  Definisce il contesto della chat, includendo lo stato, il dispatch e le funzioni per il caricamento della cronologia e l'invio dei messaggi.

```ts
import { createContext } from "react";
import { ChatAction } from "./Action";
import { ChatState } from "./ChatState";

export interface ChatContext {
  state: ChatState;
  dispatch: React.Dispatch<ChatAction>;
  loadHistory: () => Promise<void>;
  sendMessage: (text: string) => Promise<void>;
}

export const ChatContext = createContext<ChatContext | undefined>(undefined);
```

- **`ChatProviderProps.ts`**
  Specifica le proprietà richieste al provider della chat, inclusa la dipendenza dall'adapter.

```ts
import { Target } from "@/adapters/Target";
import { ReactNode } from "react";

export interface ChatProviderProps {
    children: ReactNode;
    adapter: Target;
}
```

- **`ChatState.ts`**
  Descrive lo stato della chat e definisce lo stato iniziale.

```ts
import { QuestionAnswer } from "./QuestionAnswer";

export interface ChatState {
  messages: QuestionAnswer[];
  loadingHistory: boolean;
  errorHistory: number;
  hasMore: boolean;
  hasToScroll: boolean;
}

export const initialState: ChatState = {
  messages: [],
  loadingHistory: true,
  errorHistory: 0,
  hasMore: false,
  hasToScroll: false,
};
```

- **`CustomError.ts`**
  Definisce un errore personalizzato con un codice e dettagli opzionali, migliorando la gestione e il tracciamento degli errori.

```ts
export class CustomError extends Error {
  public code: number;
  public details?: any;

  constructor(code: number, message: string, details?: any) {
```

```
        super(message);
        this.code = code;
        this.details = details;
        Object.setPrototypeOf(this, CustomError.prototype);
    }
}
```

- **`Message.ts`**
  Rappresenta un singolo messaggio con il contenuto e il timestamp.

```
export interface Message {
  content: string;
  timestamp: string;
}
```

- **`QuestionAnswer.ts`**
  Modella la struttura per una domanda e la sua risposta, includendo flag per errori e stato di caricamento

```
import { Message } from "./Message";

export interface QuestionAnswer {
    id: string;
    question: Message;
    answer: Message;
    error: number;
    loading: boolean;
    lastUpdated: string;
}
```

5.2.3) Gestione dello stato e del tema

Il frontend di BuddyBot inoltre utilizza un `Reducer` e due `Providers` che sono utilizzati per separare la logica e gestire lo stato in modo efficiente.

- Il **`ThemeProvider`** gestisce il tema visivo dell'applicazione, permettendo di applicare facilmente modalità chiare o scure (dark/light mode). Utilizzando il contesto di next-themes, consente a tutti i componenti dell'app di accedere e aggiornare dinamicamente il tema senza dover modificare manualmente ogni singolo elemento, migliorando l'esperienza utente e semplificando la gestione del design. Questo provider viene utilizzato all'interno del file `layout.tsx`.

```
"use client";

import React from "react";
import { ThemeProvider as NextThemesProvider, ThemeProviderProps } from "next-themes";

export function ThemeProvider({ children, ...props }: ThemeProviderProps) {
    return (
        <NextThemesProvider {...props}>
            {children}
        </NextThemesProvider>
    );
}
```

- Il **`ChatProvider`** è un provider che incapsula lo stato e le funzioni per gestire la chat, come il caricamento della cronologia e l'invio di messaggi. Utilizza il `useReducer` per gestire lo stato della chat, che include i messaggi, lo stato di caricamento, e gli errori. Ogni azione (come l'aggiunta di un messaggio o il caricamento della cronologia) è gestita tramite un tipo di azione definito nel reducer, che aggiorna lo stato in base al tipo di azione ricevuta. Questo provider viene utilizzato all'interno del file `ChatWindow.tsx`.

```
import React from "react";
import { useContext, useReducer, useEffect } from "react";
import { chatReducer } from "@/reducers/chatReducer";
import { initialState } from "@/types/ChatState";
import { Message } from "@/types/Message";
import { QuestionAnswer } from "@/types/QuestionAnswer";
import { generateId } from "@/utils/generateId";
import { ChatContext } from "@/types/ChatContext";
import { ChatProviderProps } from "@/types/ChatProviderProps";
import { CustomError } from "@/types/CustomError";

export const ChatProvider = ({ children, adapter }: ChatProviderProps) => {
  const [state, dispatch] = useReducer(chatReducer, initialState);

  const loadHistory = async (): Promise<void> => {
    dispatch({ type: "LOAD_HISTORY_START" });
    try {
      if (state.messages.length === 0) {
        const olderMessages: QuestionAnswer[] = await
adapter.requestHistory("", 10);
        for (let i = 0; i < olderMessages.length; i++) {
          if (olderMessages[i].answer.content.length > 100000) {
            olderMessages[i].error = 1;
          }
        }
        dispatch({ type: "LOAD_HISTORY_SUCCESS", payload: olderMessages,
hasMore: !(olderMessages.length < 10) });

        dispatch({ type: "SCROLL_DOWN" });
      }
      else {
        const olderMessages: QuestionAnswer[] = await
adapter.requestHistory(state.messages[0].id, 10);
        dispatch({ type: "LOAD_HISTORY_SUCCESS", payload: olderMessages,
hasMore: !(olderMessages.length < 10) });
      }
    }
    catch (error) {
      if (error instanceof CustomError) dispatch({ type: "LOAD_HISTORY_ERROR",
error: error.code });
      else dispatch({ type: "LOAD_HISTORY_ERROR", error: 500 });
    }
  };

  const sendMessage = async (text: string) => {
    const id = generateId();
    const newMessage: Message = {
      content: text,
      timestamp: new Date().toISOString(),
    };

    dispatch({ type: "ADD_MESSAGE_START", id: id, question: newMessage });
    dispatch({ type: "SCROLL_DOWN" });
    try {
```

```
      const botResponse: { answer: Message, id: string, lastUpdated: string } = await adapter.requestAnswer(newMessage);
      if (botResponse.answer.content.length > 100000) dispatch({ type: "ADD_MESSAGE_ERROR", id: id, error: 1 });
      else dispatch({ type: "ADD_MESSAGE_SUCCESS", id: id, answer: botResponse.answer, newid: botResponse.id, lastUpdated: botResponse.lastUpdated });

    }
    catch (error) {
      if (error instanceof CustomError) dispatch({ type: "ADD_MESSAGE_ERROR", id: id, error: error.code });
      else dispatch({ type: "ADD_MESSAGE_ERROR", id: id, error: 501 });
    }
  };

  useEffect(() => {
    loadHistory();
  }, []);

  return (
    <ChatContext.Provider value={{ state, dispatch, loadHistory, sendMessage }}>
      {children}
    </ChatContext.Provider>
  );
};

// Hook per usare il contesto
export const useChat = () => {
  const context = useContext(ChatContext);
  if (!context) {
    throw new Error("useChat must be used within a ChatProvider");
  }
  return context;
};
```

- Il **`chatReducer`** gestisce lo stato della chat, aggiornandolo in base alle azioni ricevute, come il caricamento della cronologia o l'aggiunta di nuovi messaggi. La sua struttura modulare e centralizzata consente una gestione più chiara e prevedibile dello stato, migliorando la manutenibilità e la scalabilità dell'applicazione. Separando la logica di aggiornamento dello stato dalla UI, il reducer facilita l'implementazione di nuove funzionalità senza compromettere la coerenza del sistema, rendendo l'app più facilmente estensibile e mantenibile nel tempo.

```
import { Message } from "@/types/Message";
import { ChatState } from "@/types/ChatState";
import { ChatAction } from "@/types/Action";

export const chatReducer = (state: ChatState, action: ChatAction): ChatState => {
  switch (action.type) {
    case "LOAD_HISTORY_START":
      return {
        ...state,
        loadingHistory: true,
        errorHistory: 0,
        hasMore: false,
      };
    case "LOAD_HISTORY_SUCCESS":
```

```
15        return {
16          ...state,
17          messages: [...action.payload, ...state.messages],
18          loadingHistory: false,
19          errorHistory: 0,
20          hasMore: action.hasMore,
21        };
22      case "LOAD_HISTORY_ERROR":
23        return {
24          ...state,
25          loadingHistory: false,
26          errorHistory: action.error,
27          hasMore: false,
28        };
29      case "ADD_MESSAGE_START":
30        return {
31          ...state,
            messages: [...state.messages, { id: action.id, question:
32  action.question, answer: {} as Message, error: 0, loading: true, lastUpdated:
    new Date().toISOString() }],
33        };
34      case "ADD_MESSAGE_SUCCESS":
35        const updatedMessagesSuccess = state.messages.map((msg) => {
36          if (msg.id === action.id) {
37            return {
38              ...msg,
39              id: action.newid,
40              answer: action.answer,
41              loading: false,
42              error: 0,
43              lastUpdated: action.lastUpdated,
44            };
45          }
46          return msg;
47        });
48        return {
49          ...state,
50          messages: updatedMessagesSuccess,
51        };
52      case "ADD_MESSAGE_ERROR":
53        const updatedMessagesError = state.messages.map((msg) => {
54          if (msg.id === action.id) {
55            return {
56              ...msg,
57              loading: false,
58              error: action.error,
59            };
60          }
61          return msg;
62        });
63        return {
64          ...state,
65          messages: updatedMessagesError,
66        };
67      case "SCROLL_DOWN":
68      return {
69        ...state,
70        hasToScroll: !state.hasToScroll,
71      };
72      default:
73        return state;
```

```
  }
};
```

5.2.4) Gestione e adattamento dei dati per la chat

Nel frontend di BuddyBot viene utilizzato il design pattern `Adapter` per gestire la comunicazione con le API e adattare i dati in un formato utilizzabile dall'applicazione.

- Il **`Adapter.ts`** implementa il pattern `Adapter`, che si occupa di adattare i dati ricevuti dalle API al formato richiesto dall'applicazione. Gestisce le richieste per la cronologia della chat e per ottenere le risposte alle domande, restituendo i dati come oggetti compatibili con il modello dell'app, come QuestionAnswer e Message. Le funzioni `requestHistory` e `requestAnswer` si occupano rispettivamente di recuperare la cronologia e le risposte, mentre i metodi privati all'interno dell'adapter trasformano i dati ricevuti in un formato che l'app può utilizzare facilmente.

```
import { QuestionAnswer } from "@/types/QuestionAnswer";
import { Message } from "@/types/Message";
import { Target } from "./Target";
import { Adaptee } from "./Adaptee";
import { generateId } from "@/utils/generateId";
import { CustomError } from "@/types/CustomError";

export class Adapter implements Target {
    private adaptee: Adaptee;

    constructor() {
        this.adaptee = new Adaptee();
    }

    async requestHistory(id: string, offset: number): Promise<QuestionAnswer[]>
{
        try {
            const jsonResponse = await this.adaptee.fetchHistory(id, offset);
            return this.adaptQuestionAnswerArray(jsonResponse);
        } catch (error) {
            if (error instanceof CustomError) throw error;
            throw new CustomError(500, "SERVER", "Errore interno del server");
        }
    }
    async requestAnswer(question: Message): Promise<{ answer: Message; id:
string; lastUpdated: string }> {
        try {
            const answer = await
this.adaptee.fetchQuestion(this.adaptMessageToJSON(question));
            return {
                answer: this.adaptMessage(answer.answer),
                id: answer.id,
                lastUpdated: answer.lastUpdated,
            };
        } catch (error) {
            if (error instanceof CustomError) throw error;
            throw new CustomError(501, "SERVER", "Errore interno del server");
        }
    }

    private adaptMessage(data: any): Message {
        return {
            content: data.content,
            timestamp: data.timestamp,
```

```
        };
    };

    private adaptQuestionAnswer(data: any): QuestionAnswer {
        return {
            id: data.id || generateId(),
            question: this.adaptMessage(data.question),
            answer: this.adaptMessage(data.answer),
            error: 0,
            loading: false,
            lastUpdated: data.lastUpdated,
        };
    };

    private adaptQuestionAnswerArray(dataArray: any[]): QuestionAnswer[] {
        return dataArray.map(data => this.adaptQuestionAnswer(data));
    };

    private adaptMessageToJSON(question: Message): any {
        return {
            text: question.content,
            date: question.timestamp,
        };
    };
}
```

- Il **`Adaptee.ts`** Nasconde la complessità delle chiamate di rete e fornisce metodi semplificati per ottenere la cronologia della chat e risposte alle domande, gestendo internamente i dettagli delle comunicazioni con con l'API Gateway.

```
import { CustomError } from "@/types/CustomError";

export class Adaptee {
    async fetchHistory(id: string, offset: number): Promise<any[]> {
        const controller = new AbortController();
        const timeoutId = setTimeout(() => controller.abort(), 25000);

        try {
            const response = await fetch(`http://${process.env.API_GATEWAY ?? 'localhost'}/api/get-storico?id=${id}&num=${offset}`, {
                method: "GET",
                headers: {
                    "Content-Type": "application/json",
                },
                signal: controller.signal,
            });
            clearTimeout(timeoutId);
            if (response.status >= 500) throw new CustomError(500, "SERVER", "Errore interno del server");
            if (response.status >= 400) throw new CustomError(400, "CONNESSIONE", "Errore interno del server");
            if (!response.ok) throw new CustomError(500, "SERVER", "Errore interno del server");
            return await response.json();
        } catch (error) {
            clearTimeout(timeoutId);
            if (error instanceof DOMException && error.name === "AbortError") throw new CustomError(408, "TIMEOUT", "Timeout della richiesta");
            if (error instanceof TypeError && error.message === "Failed to fetch") throw new CustomError(400, "CONNESSIONE", "Errore di connessione");
```

```
25              if (error instanceof CustomError) throw error;
26              throw new CustomError(500, "SERVER", "Errore interno del server");
27          }
28      }
29
30      async fetchQuestion(data: any): Promise<any> {
31          const controller = new AbortController();
32          const timeoutId = setTimeout(() => controller.abort(), 20000);
33
34          try {
35              const response = await fetch(`http://${process.env.API_GATEWAY ??
    'localhost'}/api/get-risposta`, {
36                  method: "POST",
37                  headers: {
38                      "Content-Type": "application/json",
39                  },
40                  body: JSON.stringify(data),
41                  signal: controller.signal,
42              });
43              clearTimeout(timeoutId);
44              if (response.status >= 500) throw new CustomError(501, "SERVER",
    "Errore interno del server");
45              if (response.status >= 400) throw new CustomError(401,
    "CONNESSIONE", "Errore interno del server");
46              if (!response.ok) throw new CustomError(501, "SERVER", "Errore
    interno del server");
47              return await response.json();
48          } catch (error) {
49              clearTimeout(timeoutId);
50              if (error instanceof DOMException && error.name === "AbortError")
    throw new CustomError(409, "TIMEOUT", "Timeout della richiesta");
51              if (error instanceof TypeError && error.message === "Failed to
    fetch") throw new CustomError(401, "CONNESSIONE", "Errore di connessione");
52              if (error instanceof CustomError) throw error;
53              throw new CustomError(501, "SERVER", "Errore interno del server");
54          }
55      }
56  }
```

5.3) Microservizio Api-Gateway

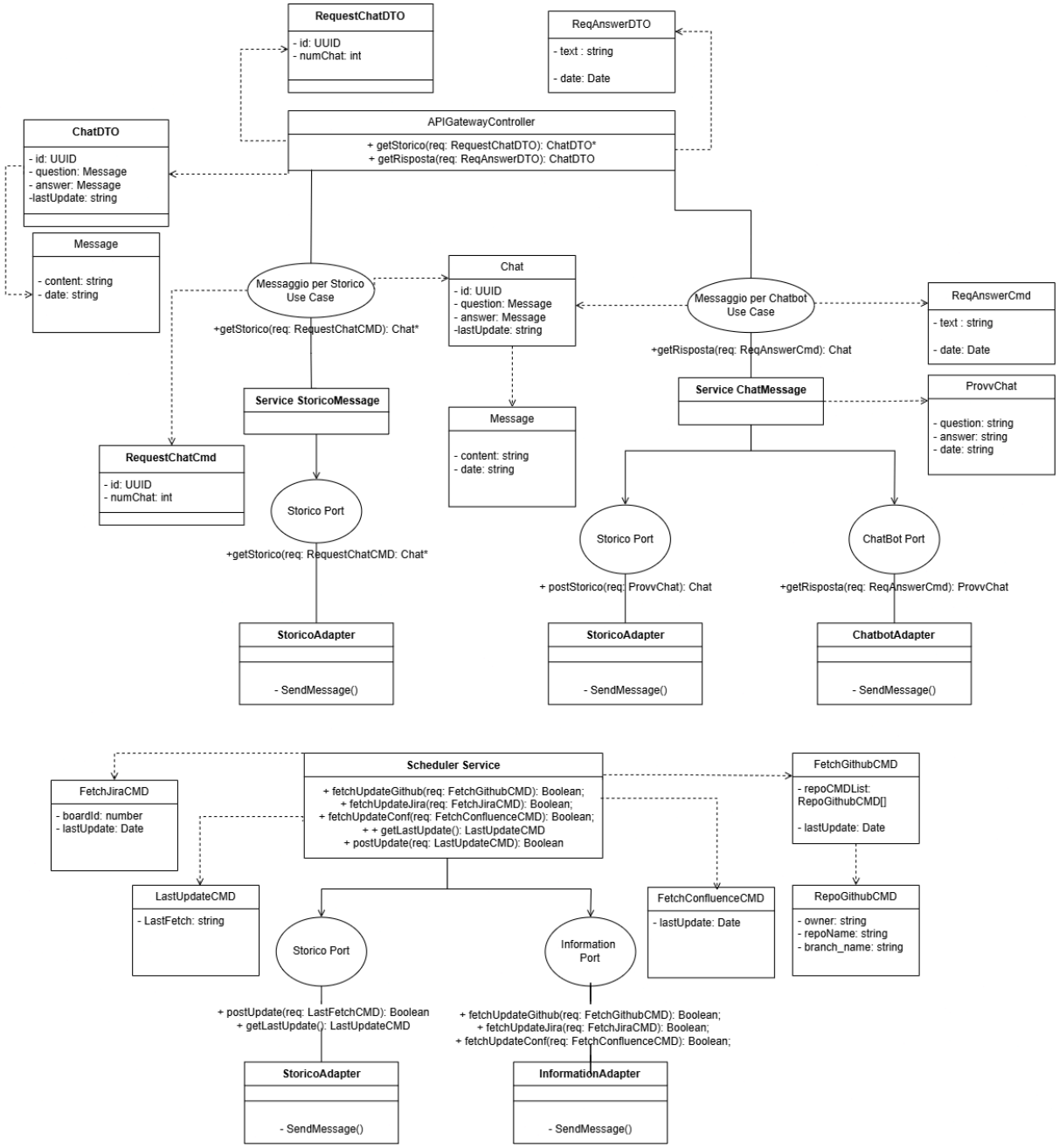

Figura 27: Diagramma UML del microservizio Api-Gateway

Il microservizio **API Gateway** svolge un ruolo cruciale nell'architettura di *BuddyBot*$_G$, fungendo da punto di ingresso centralizzato per tutte le richieste provenienti dal *front-end*$_G$ e indirizzandole verso i microservizi appropriati, garantendo il routing delle richieste e la gestione delle risposte.

Come per gli altri microservizi, anche l'API Gateway è stato progettato secondo i principi dell'architettura esagonale, al fine di garantire una netta separazione tra la logica di business e le applicazioni esterne. L'obiettivo è quello di mantenere il sistema flessibile, testabile e facilmente manutenibile.

In particolare, l'API Gateway interagisce con i microservizi tramite porte e adattatori dedicati, utilizzando *Rest-Api*$_G$ per comunicare con il *front-end*$_G$ e *RabbitMQ*$_G$ per la messaggistica con gli

altri microservizi. Questo approccio consente di mantenere l'API Gateway completamente agnostico rispetto ai dettagli di implementazione dei microservizi, favorendo una maggiore scalabilità nel futuro.

Compiti dell'API gateway:

- comunicazione attraverso *Rest-Api*$_G$ con il front-end (`@Get('get-storico')` e `@Post('get-risposta')`);
- instradamento delle richieste ai microservizi appropriati (Storico, ChatBot e Information):
  - recupero di nuova risposta dal servizio di Chatbot;
  - recupero dello storico dal servizio di Storico;
  - scheduling del fetch delle informazioni nel microservizio «Information».

5.3.1) Risposta Use-Case:

L'endpoint "get-risposta" riceve dal *front-end*$_G$ una richiesta `@Post('get-risposta')` contenente il corpo «(text)» e la data «(date)» della domanda,

```
async getRisposta(@Body('text') text: string, @Body('timestamp') timestamp: string): Promise<ChatDTO>
```

all'interno di un

```
export class ReqAnswerDTO {
    constructor(
      public readonly text: string,
      public readonly date: string
    ) {}
  }
```

e restituisce un oggetto «ChatDTO» contenente la risposta dalla domanda posta.

```
import { MessageDto } from "./message.dto";
export class ChatDTO {
  constructor(
    public readonly id: string,
    public readonly question: MessageDto,
    public readonly answer: MessageDto,
    public readonly lastUpdate: string,
  ) {}
}

export class MessageDto {
  constructor(
    public readonly content: string,
    public readonly timestamp: string,
  ) {}
}
```

Prima però la richiesta viene mandata al microservizio di «Chatbot» che restituisce una risposta

```
export class ProvChat {
  constructor(
    public readonly question: string,
    public readonly answer: string,
    public readonly timestamp: string,
  ) {}
}
```

contenente la domanda fatta e la risposta che è stata generata.

Prima di essere passata verso il front-end, «ProvChat» viene inviata al microservizio «Storico»

```
postStorico(chat: ProvChat): Promise<Chat>;
```

, il quale salva e assegna un UUID alla nuova `glossary` («Chat»), oltre alla data del `glossary` («Fetch»), in "lastUpdate" a cui appartengono le informazioni con cui è stata generata. Lo «Storico» ritorna un oggetto «Chat» completo che quindi viene passato, attraverso l' ***Endpoint***$_G$ al front-end per essere visualizzato.

5.3.2) Storico Use-Case:

Usato per caricare le chat salvate nel database del microservizio «Storico» nel front-end. L'endpoint "get-storico" riceve una richiesta all'interno di

```
  export class RequestChatDTO {
  constructor(
    public readonly id: string,
    public readonly numChat: number
  ) {}
}
```

con («id») UUID dell'ultima chat visualizzabile nell'interfaccia grafica front-end e un («numChat»), numero di chat(domanda + risposta) antecedenti a questa da caricare insieme.

```
async getStorico(@Query('id') id?: string,@Query('num') numChat?: number):
Promise<ChatDTO[]>
```

e restituisce al front-end un array di «Chat» invece che una sola. Se il sistema è stato appena avviato, viene mandata una richiesta con id =" " e num = 1 che restituisce l"ultima chat in ordine cronologico salvata nel database.

Le «Chat» recuperate con

```
getStorico(req: RequestChatCMD): Promise<Chat[]>;
```

vengono mandate al front-end con questo formato `glossary` («Json»)

```
.
.
[
      {
        "id": "ID DELLA CHAT",
        "question": {
          "content": "Domanda"
          "timestamp": "DATA DOMANDA"
        },
        "answer": {
          "content": "Risposta",
          "timestamp": "DATA RISPOSTA"
        }
      }
]
.
.
```

dove vengono suddivise e visualizzate in ordine cronologico .

5.3.3) Scheduling del Fetch:

Inoltre Api-Gateway si occupa anche dello scheduling del fetch delle informazioni nel microservizio di «Information» e del passaggio della data in cui viene effettuato al microservizio di «Storico» con

```
postUpdate(LastFetch:string): Promise<Boolean>;
```

per essere salvata e poi fornita all'utente all'interno della `glossary` («Chat») che riceve indicando a quando risalgono le informazioni usate per formulare la risposta.

Prima però viene fatto un check per controllare se esiste una data di $fetch_G$ nel database con

```
getLastUpdate(): Promise<LastUpdateCMD>;
```

, se non esiste significa che non è stato ancora fatto nessun fetch e in questo caso viene effettuato un fetch completo che recupera tutte le informazioni. In questo caso noi abbiamo messo la data di qualche mese fa per facilitare il test siccome il fetch, soprattutto di github, richiede tempo, ma se non si mettesse una data viene fatto il fetch di tutto.

Nel caso invece esista questa viene usata come data di partenza.

Per gestire lo scheduling viene usato `@Cron` della libreria `@nestjs/schedule`(in questo caso è stato impostato per essere effettuato ogni 5 minuti su richiesta dell'azienda). Oltre alla data vengono passati anche una serie di oggetti che contengono dati sulle repository che vengono usati dal microservizio di «Information» per fare il fetch delle informazioni.

```
  ...
export class TasksService implements OnModuleInit {
  private readonly logger = new Logger(TasksService.name);

  constructor(
    @Inject('InfoPort') private readonly infoPort: InfoPort,
    @Inject('StoricoPort') private readonly storicoPort: StoricoPort,
  ) {}

  @Cron('0 */5 * * * *')
  async handleCron() {
    this.logger.debug('Esecuzione FETCH ogni TOT (ogni 5 min)...');
    await this.runFetch();
  }

  private async runFetch() {
    try {
      this.logger.debug('Richiesta della data di ultimo FETCH (SERVICE)');
      const isoDateString = await this.storicoPort.getLastUpdate();

      let DataFetch: Date;

      if (!isoDateString?.LastFetch) {

        DataFetch = new Date();
        DataFetch.setMonth(DataFetch.getMonth() - 9);
        this.logger.warn(`Nessuna data FETCH (SERVICE) precedente. Uso data di
fallback: ${DataFetch}`);
      } else {
        DataFetch = new Date(isoDateString.LastFetch);
```

```
          this.logger.debug(`FETCH (SERVICE) da data salvata trovata: ${DataFetch}
`);
        }

        const boardId = 1;
        const jiraCmd = new FetchJiraCMD(boardId, DataFetch);
        const confCmd = new FetchConfluenceCMD(DataFetch);

        const owner = process.env.GITHUB_OWNER || 'SweeTenTeam';
        const repoName = process.env.GITHUB_REPO || 'BuddyBot';
        const branch = process.env.GITHUB_BRANCH || 'develop';
        const repoCMD = new RepoGithubCMD(owner, repoName, branch);
        const githubCmd = new FetchGithubCMD([repoCMD], DataFetch);

        const resultFetchJira = await this.infoPort.fetchUpdateJira(jiraCmd);
        const resultFetchConf = await this.infoPort.fetchUpdateConf(confCmd);
        const resultFetchGithub = await
this.infoPort.fetchUpdateGithub(githubCmd);

        if (resultFetchJira && resultFetchGithub && resultFetchConf) {
          this.logger.log(`FETCH (SERVICE) completato con successo.`);

          const NewDataFetch = new Date();
          const lastUpdateCmd = new LastUpdateCMD(NewDataFetch.toISOString());
          const result = await this.storicoPort.postUpdate(lastUpdateCmd);

          this.logger.debug(`Salvataggio data fetch riuscito: ${result}`);
        } else {
          this.logger.error(`FETCH (SERVICE) fallito: almeno uno dei servizi ha
dato errore.`);
        }
      } catch (error) {
        this.logger.error('Errore nel FETCH (SERVICE) iniziale', error);
      }
    }
}
```

Con **“FetchGithubCMD”** che contiene le informazioni della repo a cui fare riferimento, questi sono salvati in un file “.env” per essere facilmente modificabili.

```
import { RepoGithubCMD } from "./RepoGithubCMD.js";
export class FetchGithubCMD {
    constructor (
        public readonly repoDTOList: RepoGithubCMD[],
        public readonly lastUpdate: Date
    ){}
}
//CHE USA
export class RepoGithubCMD{
    constructor(
        public readonly owner: string,
        public readonly repoName: string,
        public readonly branch_name: string
    ){}
}
```

Sono state messe 3 diverse funzioni per il fetch , una per ogni fonte, per rendere il codice facilmente espandibile in futuro, nel caso si vogliano aggiungere nuovi fonti basterà aggiungere la loro funzione e creare il loro oggetto con i dati necessari. Ma anche nel caso si voglia dare tempi di scheduling differenti ad ogni fonte e salvare nel database date di diverse per ciascuna.

```
export interface InfoPort {
  fetchUpdateGithub(req: FetchGithubCMD): Promise<Boolean>;
  fetchUpdateJira(req: FetchJiraCMD): Promise<Boolean>;
  fetchUpdateConf(req: FetchConfluenceCMD): Promise<Boolean>;
}
```

5.4) Microservizio Chatbot

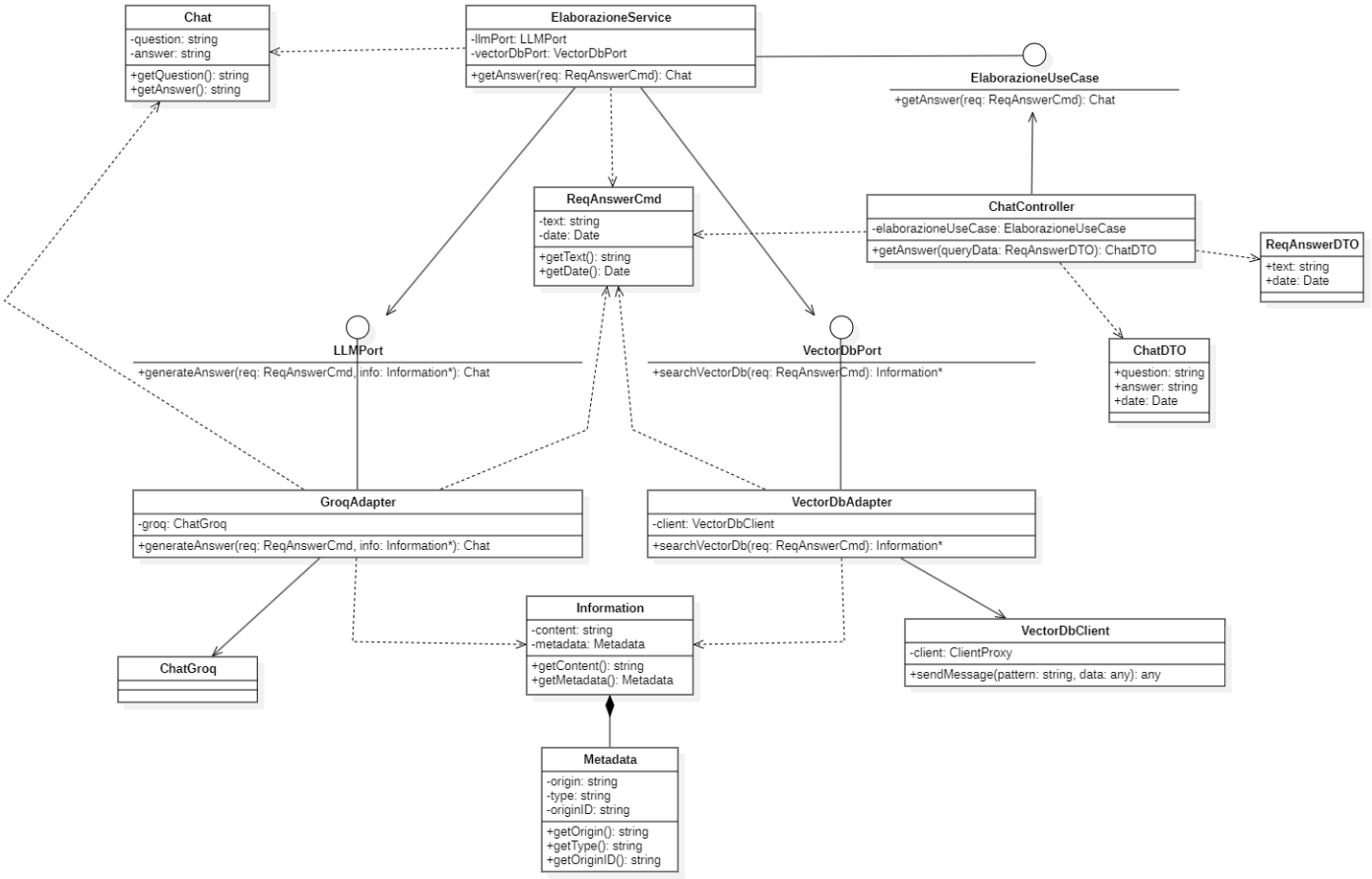

Figura 28: UML ChatBot

Il microservizio Chatbot rappresenta una componente cruciale all'interno dell'architettura di *Buddy-Bot*$_G$, essendo responsabile dell'elaborazione delle domande degli utenti e della generazione di risposte pertinenti. Questo microservizio è progettato secondo i principi dell'architettura esagonale garantendo una netta separazione tra la logica di business e i dettagli implementativi.

La sua funzione principale è quella di ricevere una domanda dall'utente, arricchirla con informazioni contestuali recuperate dal microservizio Informazioni, e utilizzare queste informazioni per generare una risposta accurata e rilevante attraverso un modello di linguaggio esterno (LLM).

5.4.1) Architettura e Componenti

L'architettura del microservizio è strutturata in diversi layer, ciascuno con responsabilità ben definite:

5.4.1.1) Domain Layer

Il Domain Layer contiene le entità core e i value objects che rappresentano i concetti fondamentali del dominio, indipendenti da qualsiasi tecnologia specifica:

- **Entità**:
  - `Chat`: Rappresenta una conversazione completa con domanda e risposta
  - `Information`: Contiene informazioni contestuali recuperate dal database vettoriale
  - `Metadata`: Mantiene i metadati associati alle informazioni (origine, tipo, ID)
- **Value Objects**:
  - `ReqAnswerCmd`: Command object che incapsula la richiesta dell'utente

5.4.1.2) Application Layer

L'Application Layer coordina il flusso di dati e implementa i casi d'uso dell'applicazione, orchestrando il lavoro delle entità del dominio:

- **Use Cases (Interfaces)**:
  - `ElaborazioneUseCase`: Definisce il contratto per l'elaborazione delle domande e la generazione di risposte
- **Ports (Interfaces)**:
  - `LLMPort`: Interfaccia che definisce le operazioni per interagire con modelli di linguaggio esterni
  - `VectorDbPort`: Interfaccia che definisce le operazioni per recuperare informazioni dal database vettoriale
- **Services**:
  - `ElaborazioneService`: Implementazione concreta di `ElaborazioneUseCase` che coordina l'interazione tra il recupero delle informazioni contestuali e la generazione delle risposte attraverso il modello di linguaggio

Questo layer implementa la logica applicativa senza dipendere direttamente da meccanismi specifici di persistenza o comunicazione, utilizzando le interfacce (ports) per interagire con il mondo esterno.

5.4.1.3) Adapters Layer

L'Adapters Layer traduce le interazioni tra il core dell'applicazione e il mondo esterno, gestendo le conversioni di formato e protocollo:

- **Adapters In**:
  - `ChatController`: Riceve le richieste tramite RabbitMQ, le converte in command objects (`ReqAnswerCmd`) e le passa al caso d'uso appropriato (`ElaborazioneUseCase`)
- **Adapters Out**:
  - `GroqAdapter`: Implementa `LLMPort` per interagire con il modello di linguaggio *Groq*[G], convertendo i formati di dominio in richieste API specifiche
  - `VectorDbAdapter`: Implementa `VectorDbPort` per comunicare con il microservizio DB Vettoriale, gestendo la serializzazione e deserializzazione dei messaggi RabbitMQ
- **Data Transfer Objects (DTOs)**:
  - `ReqAnswerDTO`: Oggetto di trasferimento dati per ricevere le richieste in ingresso dal client
  - `ChatDTO`: Oggetto di trasferimento dati per le risposte (definito ma non utilizzato nell'implementazione attuale)

Gli adapter isolano il core dell'applicazione dai dettagli di implementazione delle tecnologie esterne, consentendo di sostituire facilmente tali tecnologie senza modificare la logica di business.

5.4.1.4) Infrastructure Layer

L'Infrastructure Layer fornisce implementazioni concrete per servizi esterni, configurazioni e meccanismi di comunicazione:

- **Clients**:
  - `VectorDbClient`: Client che gestisce la comunicazione con il microservizio DB Vettoriale tramite RabbitMQ, incapsulando i dettagli di connessione e serializzazione
  - `ChatGroq`: Client di terze parti per l'interazione con l'API di *Groq*$_G$, configurato per utilizzare il modello di linguaggio «qwen-2.5-32b»
- **Configuration**:
  - `ConfigModule`: Modulo di NestJS che gestisce il caricamento e l'accesso alle variabili d'ambiente
  - `AppModule`: Modulo principale dell'applicazione che configura le dipendenze, i provider e i controller
- **Communication**:
  - `rabbitMQConfig`: Configurazione per la connessione a RabbitMQ, definendo code e opzioni

Questo layer si concentra esclusivamente sui dettagli tecnici e sulle implementazioni specifiche delle tecnologie, mantenendo queste preoccupazioni separate dalla logica di business.

5.4.2) Flusso Principale di Elaborazione

Il flusso principale per la generazione di una risposta segue questi passaggi:

1. **Ricezione della richiesta**
   - Un messaggio contenente la domanda dell'utente viene ricevuto tramite RabbitMQ
   - Il `ChatController` gestisce il messaggio e crea un comando `ReqAnswerCmd`
2. **Ricerca di informazioni contestuali**
   - Il servizio `ElaborazioneService` utilizza `VectorDbPort` per cercare informazioni rilevanti nel *database vettoriale*$_G$
   - La richiesta viene inoltrata al microservizio Informazioni tramite RabbitMQ
3. **Generazione della risposta**
   - Le informazioni contestuali recuperate vengono combinate con la domanda originale
   - Il servizio utilizza `LLMPort` per interagire con un modello di linguaggio (*Groq*$_G$)
   - La risposta generata viene formattata come oggetto `Chat`

4. **Restituzione della risposta**
   - Il risultato viene restituito al chiamante (API Gateway)

5.4.3) Componenti Principali

5.4.3.1) Controllers

- **ChatController**: Punto di ingresso per le richieste RabbitMQ. Gestisce il pattern di messaggistica «get-answer» e converte i dati di richiesta (`ReqAnswerDTO`) in comandi di dominio (`ReqAnswerCmd`).

5.4.3.2) Use Cases e Ports

- **ElaborazioneUseCase**: Interfaccia che definisce il contratto per il caso d'uso principale di generazione di risposte.
- **LLMPort**: Interfaccia che definisce il contratto per l'interazione con modelli di linguaggio.
- **VectorDbPort**: Interfaccia che definisce il contratto per l'interazione con il *database vettoriale*[G].

5.4.3.3) Services

- **ElaborazioneService**: Implementazione principale del caso d'uso di elaborazione delle domande. Gestisce il flusso complessivo dell'elaborazione della richiesta:
  1. Ricerca di informazioni contestuali rilevanti tramite `VectorDbPort`
  2. Invio della domanda e del contesto al modello di linguaggio tramite `LLMPort`
  3. Restituzione della risposta generata

```
@Injectable()
export class ElaborazioneService implements ElaborazioneUseCase {
  constructor(
    @Inject(LLM_PORT)
    private readonly llmPort: LLMPort,
    @Inject(VECTOR_DB_PORT)
    private readonly vectorDbPort: VectorDbPort,
  ) {}

  async getAnswer(req: ReqAnswerCmd): Promise<Chat> {
      // 1. Ricerca del contesto rilevante nel database vettoriale tramite
RabbitMQ
      const relevantContext = await this.vectorDbPort.searchVectorDb(req);
      console.log(`Retrieved ${relevantContext.length} relevant documents: `);

      // 2. Genera la risposta utilizzando l'LLM con il contesto recuperato
      const chat = await this.llmPort.generateAnswer(req, relevantContext);

      return chat;
  }
}
```

5.4.3.4) Adapters

- **GroqAdapter**: Implementa `LLMPort` per interagire con il modello di linguaggio *Groq*$_G$. Utilizza *LangChain*$_G$ per gestire i prompt e il parsing delle risposte.

```
@Injectable()
export class GroqAdapter implements LLMPort {
  constructor(private readonly groq: ChatGroq) {

  }

  async generateAnswer(req: ReqAnswerCmd, info: Information[]): Promise<Chat> {
    const prompt = PromptTemplate.fromTemplate(`Answer the question based only on the following context: {context} Question: {question}`);
    const ragChain = await createStuffDocumentsChain({
        llm: this.groq,
        prompt,
        outputParser: new StringOutputParser(),
    });
    const documents: Document[] = [];
    for(const information of info){
      documents.push({
        pageContent: information.content,
        metadata: {
          'origin': information.metadata.origin,
          'type': information.metadata.type,
          'originId': information.metadata.originID
        }
      });
    }
    const response = await ragChain.invoke({
      question: req.getText(),
      context: documents
    });
    return new Chat(req.getText(), response);
  }
}
```

- **VectorDbAdapter**: Implementa `VectorDbPort` per interagire con il microservizio DB Vettoriale tramite RabbitMQ.

```
@Injectable()
export class VectorDbAdapter implements VectorDbPort {
    constructor(private client: VectorDbClient) {}

    async searchVectorDb(req: ReqAnswerCmd): Promise<Information[]> {
        let result: Information[] = [];
        const res = await this.client.sendMessage("retrieve.information", {query: req.getText()});

        for(const r of JSON.parse(JSON.stringify(res))) {
            let i = new Information(
                r.content,
                new Metadata(r.metadata.origin, r.metadata.type, r.metadata.originID)
            );
            result.push(i);
        }

        return result;
    }
}
```

5.4.3.5) Entità e Value Objects

- **Chat**: Rappresenta una conversazione completa, contenente sia la domanda che la risposta.

```
export class Chat {
  private question: string;
  private answer: string;

  constructor(question: string, answer: string) {
    this.question = question;
    this.answer = answer;
  }

  getQuestion(): string {
    return this.question;
  }

  getAnswer(): string {
    return this.answer;
  }
}
```

- **Information**: Rappresenta le informazioni contestuali recuperate dal *database vettoriale*$_G$.

```
export class Information {
  constructor(
    public readonly content: string,
    public readonly metadata: Metadata,
  ){}

  getContent(): string {
    return this.content;
  }

  getMetadata(): Metadata {
    return this.metadata;
  }
}
```

- **Metadata**: Contiene metadati associati alle informazioni contestuali.

```
export class Metadata {
  constructor(
    public readonly origin: string,
    public readonly type: string,
    public readonly originID: string,
  ) {}

  getOrigin(): string {
    return this.origin;
  }

  getType(): string {
    return this.type;
  }

  getOriginID(): string {
    return this.originID;
  }
}
```

5.4.4) Integrazione con LangChain e Groq

Il microservizio utilizza *LangChain*$_G$ come framework per semplificare l'interazione con i modelli di linguaggio. In particolare:

1. **Costruzione dei Prompt**: Utilizza `PromptTemplate` per strutturare i prompt con un formato coerente.
2. **Catene di Elaborazione**: Utilizza `createStuffDocumentsChain` per combinare documenti di contesto con la domanda dell'utente.
3. **Parsing delle Risposte**: Utilizza `StringOutputParser` per estrarre il testo dalla risposta del modello.

Per l'integrazione con il modello *Groq*$_G$, il servizio utilizza il modello «qwen-2.5-32b» con i seguenti parametri:

- Limite di token: 6000
- Numero massimo di tentativi: 2

```
{
  provide: ChatGroq,
  useFactory: () => {
    return new ChatGroq({
      apiKey: process.env.GROQ_API_KEY,
      model: "qwen-2.5-32b",
      maxTokens: 6000,
      maxRetries: 2,
    });
  },
}
```

5.4.5) Comunicazione con Altri Microservizi

La comunicazione con altri microservizi avviene principalmente tramite RabbitMQ:

1. **Ricezione di Richieste dall'API Gateway**:
   - Coda: «chatbot-queue»
   - Pattern di messaggistica: «get-answer»
   - Payload: `ReqAnswerDTO` contenente il testo della domanda e il timestamp
2. **Invio di Richieste al DB Vettoriale**:
   - Coda: «information-queue»
   - Pattern di messaggistica: «retrieve.information»
   - Payload: Oggetto contenente la query da cercare

La configurazione RabbitMQ è definita nel file `main.ts`:

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>(
  AppModule,
  {
    transport: Transport.RMQ,
    options: {
      urls: [process.env.RABBITMQ_URL || 'amqp://rabbitmq'],
      queue: 'chatbot-queue',
      queueOptions: {
        durable: true,
      },
    },
  },
);
```

5.4.6) Configurazione e Ambiente

Il microservizio utilizza variabili d'ambiente per gestire le configurazioni:

- `RABBITMQ_URL`: URL del server RabbitMQ (default: `amqp://rabbitmq`)
- `GROQ_API_KEY`: Chiave API per l'accesso al servizio Groq

La configurazione dell'ambiente è gestita tramite il modulo `ConfigModule` di NestJS, che carica automaticamente le variabili d'ambiente all'avvio dell'applicazione.

5.4.7) Conclusione

Il microservizio Chatbot rappresenta il cuore intelligente di ***BuddyBot***$_G$, responsabile della generazione di risposte accurate e contestualmente rilevanti. La sua architettura esagonale garantisce una chiara separazione delle responsabilità, facilitando la manutenzione e l'evoluzione del sistema nel tempo. L'integrazione con ***LangChain***$_G$ e ***Groq***$_G$ fornisce capacità avanzate di elaborazione del linguaggio naturale, mentre la comunicazione tramite RabbitMQ assicura un'integrazione efficiente con gli altri componenti del sistema.

5.5) Microservizio Storico Chat

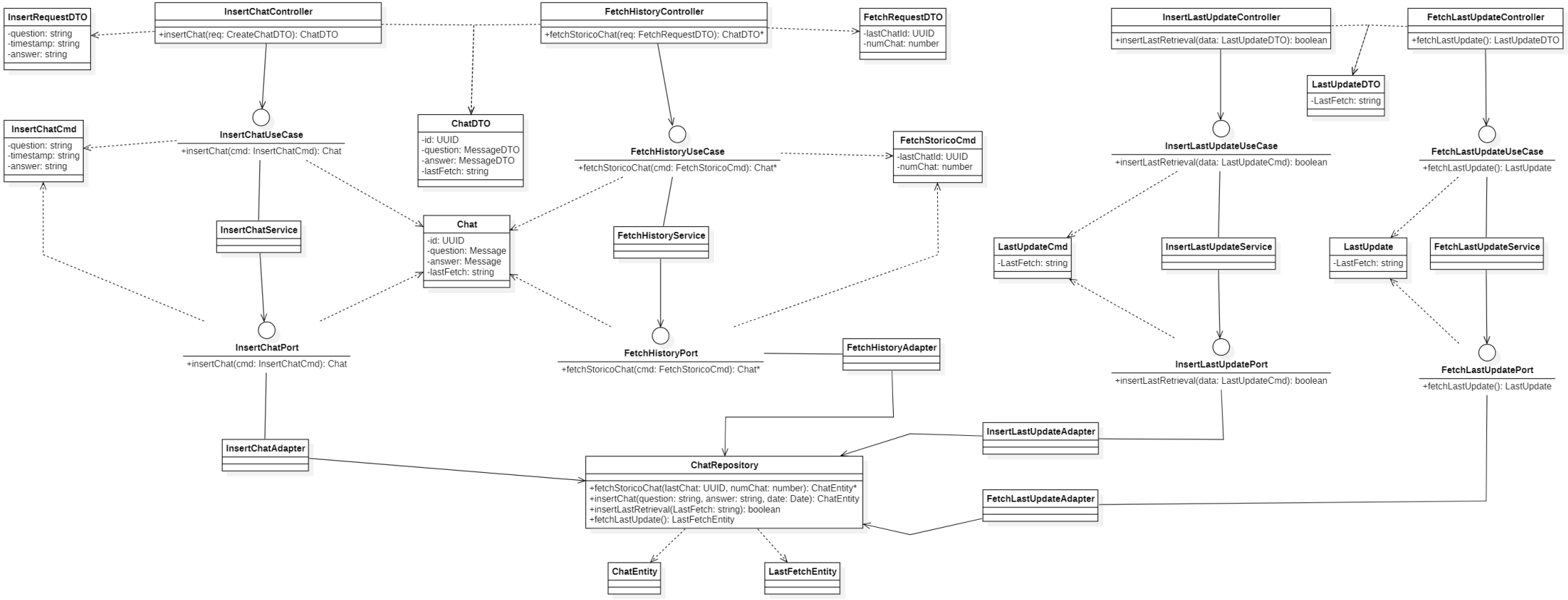

Figura 29: Progettazione del Microservizio Storico Chat

Il microservizio dello Storico riveste un ruolo fondamentale per il corretto funzionamento di *BuddyBot*$_G$: esso si occupa della gestione delle interazioni con il database relazionale *PostgreSQL*$_G$, prelevando e inserendo dati relativi alle conversazioni in modo affidabile. Come per gli altri microservizi, anche questo è stato progettato secondo i principi dell'architettura esagonale, al fine di garantire una netta separazione tra la logica di business e i dettagli di implementazione tecnologica. In particolare, l'interazione con PostgreSQL è delegata a un repository dedicato (`ChatRepository`), che utilizza *TypeORM*$_G$ per l'accesso e la gestione delle entità persistite. La logica applicativa, invece, accede ai dati attraverso alle Port & Adapter di output, fungendo da mediatori con il repository. Questo approccio consente di mantenere l'«application» completamente agnostica rispetto alla tecnologia di persistenza, favorendo una maggiore manutenibilità, testabilità e flessibilità.

5.5.1) Quattro casi d'uso

Questo microservizio è stato progettato per l'esecuzione di 4 principali operazioni.

- **Recupero dello Storico della Chat**
  - L'obiettivo è quello di recuperare dal database una specifica quantità di messaggi richiesti
- **Inserimento di nuovi messaggi**
  - L'obiettivo è quello di inserire nuovi messaggi presenti nella *UI*$_G$ nel database, in maniera tale da permettere successivi recuperi
- **Inserimento dell'ultima data di recupero informazioni** (*Retrieval Periodico*$_G$)
  - Il sistema esegue un recupero periodico dei dati provenienti da *Jira*$_G$, *GitHub*$_G$, *Confluence*$_G$. In questo microservizio si vuole memorizzare l'ultima data di recupero nel database (sovrascrivendo quella precedente se presente), così da poterla restituire insieme ai dati della chat.
- **Ottenimento della data di ultimo recupero / aggiornamento informazioni**
  - L'obiettivo è quello di recuperare correttamente nella tabella dedicata l'unico record presente rappresentante la data in cui è stato eseguito l'ultimo *Retrieval Periodico*$_G$.

Nelle prossime sezioni verranno riepilogati i 4 flussi per le rispettive operazioni.

5.5.2) Recupero dello Storico della Chat

- **`FetchRequestDTO`**: rappresenta il Data Transfer Object utilizzato per contenere la richiesta di recupero dello storico. Include due parametri, ovvero:
  - ID: identificativo che rappresenta l'ultima `Chat` (coppia di messaggi, come verrà spiegato nella specifica classe) precedentemente caricata. Questo valore viene utilizzato come punto di riferimento cronologico per effettuare il fetch dei messaggi successivi, seguendo un ordinamento decrescente (dal più recente al meno recente);
  - numChat: quantità delle chat che si vogliono recuperare nella medesima operazione.

  Il DTO in questo caso è essenziale per permettere un corretto trasferimento dei dati tra microservizi e livelli differenti.

- **`FetchHistoryController`**: corrisponde al consumer, rimane in ascolto nella coda "fetch_queue" e in ricezione ottiene un messaggio contenente una richiesta presente in un oggetto DTO - `FetchRequestDTO`. Il controller si occupa di trasformare il DTO in un oggetto `FetchHistoryCmd`, delegando l'elaborazione allo *UseCase* (interface) e alla sua corrispettiva implementazione, ossia al *Service*. Una volta che quest'ultimo tornerà l'oggetto di dominio, il controller lo convertirà nuovamente in ChatDTO affinchè vengano rispettati i principi del modello esagonale.

- **`FetchHistoryCmd`**: command object creato a partire dal DTO, formalizza e incapsula i parametri effettivi della richiesta. Utile a separare i dati provenienti dall'esterno dal formato atteso dalla logica di business, garantendo isolamento tra livelli. I parametri presenti all'interno di tale richiesta sono sempre "ID" e "numChat", citati e spiegati in precedenza per il `FetchRequestDTO`.

- **`FetchHistoryUseCase`**: interfaccia che rappresenta la porta di ingresso della logica applicativa per il recupero dello storico. Utile per garantire disaccopiamento tra *Controller* e *Service*. Nel metodo esposto per il recupero viene passsato `FetchHistoryCmd` come input, mentre in output si ritorna l'oggetto di dominio, ossia `Chat`.
- **`FetchHistoryService`**: implementazione concreta dell'interfaccia precedente, è la classe principale della business logic. Non interagisce direttamente con il database, il suo ruolo è quello di **orchestrare** un corretto recupero dello storico presente in database. In linea con i principi dell'architettura esagonale, questa classe consente di mantenere la logica di business indipendente dall'infrastruttura.
- **`FetchHistoryPort`**: questa interfaccia rappresenta la porta di uscita (output port) dal punto di vista della logica applicativa. Astrae il meccanismo con cui vengono recuperati i dati dal livello di persistenza.
- **`FetchHistoryAdapter`**: implementazione concreta dell'interfaccia spiegata in precedenza, funge da punto di collegamento tra logica applicativa al sistema di persistenza ma non accede al database. Il suo primo compito è quello di formalizzare la richiesta ricevuta (`FetchHistoryCmd`) in un formato adatto al repository, estraendo e passando in modo esplicito i parametri (id, numChat) necessari alla query. Dopodichè riceve i dati persistiti (`ChatEntity`), li trasforma in dati di dominio (`Chat`) e li restituisce al *Service*.
- **`ChatRepository`**: È la componente incaricata dell'accesso diretto a PostgreSQL, utilizzando *TypeORM* per la gestione delle entità e delle query. Fornisce il metodo fetchStoricoChat, che implementa la logica di recupero dei messaggi in due scenari distinti:
  - nel caso di primo accesso a BuddyBot (quando non è fornito un id), vengono recuperate le conversazioni più recenti, ordinate per data in modo decrescente;
  - nei casi successivi, viene prima identificata la chat corrispondente all'id fornito e, a partire dalla sua data, vengono recuperate le conversazioni precedenti.

  A seguire, viene inserito il metodo «fetchStoricoChat()» presente in questa classe.

```
...
export class ChatRepository {
  constructor(
    @InjectRepository(ChatEntity) //tabella db della chat
    private readonly chatRepo: Repository<ChatEntity>,

    @InjectRepository(LastUpdateEntity) //tabella db con unico record data ultimo retrieval info
    private readonly lastUpdateRepo: Repository<LastUpdateEntity>,
  ) {}


  async fetchStoricoChat(lastChatId: string, numChat?: number): Promise<ChatEntity[]> {
  try {
    const take = numChat ? numChat : 5;

    //caso senza ID (quindi primo accesso)
    if (!lastChatId) {
      const lastChats = await this.chatRepo.find({
        order: { answerDate: 'DESC' },
        take,
      });
      return lastChats.slice().reverse()
    }

```

```
      //caso con ID, trovo chat corrispondente e prendo le precedenti (ragionando in ordine cronologico)
      const lastChat = await this.chatRepo.findOne({
        where: { id: lastChatId },
      });

      if (!lastChat) {
        throw new Error('Last chat ID not found');
      }

      const previousChats = await this.chatRepo.find({
        where: {
          answerDate: LessThan(lastChat.answerDate),
        },
        order: { answerDate: 'DESC' },
        take: take,
      });
      const combo = previousChats.slice().reverse()
      return combo;

    } catch (error) {
      console.error('Error during History-fetch:', error);
      throw new Error('Error during History-fetch');
    }
    ...
  }
```

- **`Chat`**: rappresenta l'entità di dominio; una singola Chat rappresenta una **coppia di messaggi**, ossia include una domanda e la rispettiva risposta. La conversazione con *BuddyBot*$_G$, quindi, si compone di Chats.

```
export class Chat {
  constructor(
    public readonly id: string,
    public readonly question: Message,
    public readonly answer: Message,
    public readonly lastFetch: string
  ) {}
}
```

- **`ChatDTO`**: data transfer object di uscita, costruito dal controller a partire dagli oggetti Chat.
- **`Message`**: rappresenta l'entità di dominio che incapsula le informazioni relative a un singolo messaggio all'interno di una `Chat`.

```
  export class MessageDTO {
  constructor(
    public readonly content: string,
    public readonly timestamp: string,
  ) {}
}
```

- **`MessageDTO`**: data transfer object utilizzato per esporre i singoli messaggi all'esterno.
- **`ChatEntity`**: rappresenta la mappatura dell'entità «Chat» nel database PostgreSQL, gestita tramite *TypeORM*$_G$. E' associata a una tabella generata automaticamente e viene utilizzata per persistere ogni conversazione tra l'utente e BuddyBot. I principali campi della classe sono:
  - id: chiave primaria generata in formato UUID;

- question: il contenuto testuale della domanda posta dall'utente;
- questionDate: timestamp associato alla domanda. Il valore di questo campo viene esplicitamente passato tramite la richiesta di inserimento e conservato così com'è nel database;
- answer: il contenuto testuale della risposta generata;
- answerDate: a differenza della *questionDate*, è un timestamp generato automaticamente al momento dell'inserimento nel database. È gestito da TypeORM tramite il decoratore @CreateDateColumn, che assegna il valore corrente (now) senza necessità di specificarlo a livello applicativo.
- lastFetch: rappresenta la data dell'ultimo «*Retrieval Periodico*$_G$» eseguito, dando all'utilizzatore la possibilità di capire quanto recenti (o meno) sono i dati elaborati dal chatbot.

```
import { Column, CreateDateColumn, Entity, PrimaryGeneratedColumn } from
"typeorm";

@Entity()
export class ChatEntity {
    @PrimaryGeneratedColumn('uuid') //primaryKey
    id: string;

    @Column()
    question: string;

    @Column({ type: 'timestamptz' })
    questionDate: Date;

    @Column()
    answer: string;

    @CreateDateColumn({ type: 'timestamptz', default: () =>
'CURRENT_TIMESTAMP' })
    answerDate: Date = new Date();

    @Column()
    lastFetch: string;
}
```

5.5.3) Inserimento di nuovi messaggi

- **`InsertRequestDTO`**: rappresenta il Data Transfer Object utilizzato per contenere la richiesta di inserimento nel database di una nuova Chat (coppia di messaggi). Include tre parametri, ovvero:
  - question: una stringa contenente la domanda posta;
  - timestamp: una stringa contenente la data+orario dell'invio della domanda
    - si osservi che viene passata solamente quella domanda poiché quella della risposta viene decretata una volta avvenuto l'inserimento in database;
  - answer: una stringa contenente la risposta generata dal chatbot.

  Il DTO in questo caso è essenziale per permettere un corretto traferimento dei dati tra microservizi e livelli differenti.

- **`InsertChatController`**: corrisponde al consumer, rimane in ascolto nella coda "chat_message" e in ricezione ottiene un messaggio contenente una richiesta presente in un oggetto DTO - `InsertRequestDTO`. Il controller si occupa di trasformare il DTO in un oggetto `InsertChatCmd`, delegando l'elaborazione allo *UseCase* (interface) e alla sua corrispettiva implementazione, ossia al *Service*. Una volta che quest'ultimo tornerà l'oggetto di dominio, il controller lo convertirà nuovamente in ChatDTO affinchè vengano rispettati i principi del modello esagonale.

- **`InsertChatCmd`**: command object creato a partire dal DTO, formalizza e incapsula i parametri effettivi della richiesta. Utile a separare i dati provenienti dall'esterno dal formato atteso dalla logica di busi-

ness, garantendo isolamento tra livelli. I parametri presenti all'interno di tale richiesta rimangono i medesimi citati e spiegati in precedenza per il `InsertRequestDTO`.

- **`InsertChatUseCase`**: interfaccia che rappresenta la porta di ingresso della logica applicativa per l'inserimento in database di una nuova `Chat`. Utile per garantire disaccopiamento tra *Controller* e *Service*.
- **`InsertChatService`**: implementazione concreta dell'interfaccia precedente, è la classe principale della business logic. Non interagisce direttamente con il database, il suo ruolo è quello di **orchestrare** un corretto inserimento di una `Chat` nel database. In linea con i principi dell'architettura esagonale, questa classe consente di mantenere la logica di business indipendente dall'infrastruttura.
- **`InsertChatPort`**: questa interfaccia rappresenta la porta di uscita (output port) dal punto di vista della logica applicativa. Astrae il meccanismo mediante il quale viene eseguito il processo di inserimento dati nel database.
- **`InsertChatAdapter`**: implementazione concreta dell'interfaccia spiegata in precedenza, funge da punto di collegamento tra logica applicativa al sistema di persistenza ma non accede al database. Il suo primo compito è quello di formalizzare la richiesta ricevuta (`FetchHistoryCmd`) in un formato adatto al repository, estraendo e passando in modo esplicito i parametri (id, numChat) necessari alla query. Dopodichè riceve i dati persistiti (`ChatEntity`), li trasforma in dati di dominio (`Chat`) e li restituisce al *Service*.
- **`ChatRepository`**: componente incaricata dell'accesso diretto a PostgreSQL, utilizzando *TypeORM*$_G$ per la gestione delle entità e delle query. Fornisce il metodo insertStoricoChat(), che ha il compito di persistere una nuova conversazione nel database. Prima di creare la nuova entità, viene effettuata una lettura dal repository `lastUpdateRepo`, per recuperare il valore corrente dell'ultima data di aggiornamento, *lastFetch*, utilizzato poi per popolare il medesimo campo della nuova conversazione (domanda-risposta).

  A seguire, viene inserito il metodo «insertStoricoChat()» presente in questa classe.

```
...
export class ChatRepository {
  constructor(
    @InjectRepository(ChatEntity) //tabella db della chat
    private readonly chatRepo: Repository<ChatEntity>,

    @InjectRepository(LastUpdateEntity) //tabella db con unico record data ultimo retrieval info
    private readonly lastUpdateRepo: Repository<LastUpdateEntity>,
  ) {}


  async insertChat(question: string, answer: string, date: Date): Promise<ChatEntity> {
    const lastUpdate = await this.lastUpdateRepo.findOne({ where: { id: 1 } });

    if (!lastUpdate) {
      throw new Error('LastUpdate entry not found');
    }

    const newChat: ChatEntity = this.chatRepo.create({
      question,
      questionDate: date,
      answer,
      lastFetch: lastUpdate.lastFetch.toISOString()
    });
```

```
25
26      await this.chatRepo.save(newChat);
27
28      return newChat;
29    }
30    ...
31  }
```

- **`Chat`**: rappresenta l'entità di dominio; una singola Chat rappresenta una **coppia di messaggi**, ossia include una domanda e la rispettiva risposta. La conversazione con *BuddyBot*$_G$, quindi, si compone di Chats.
- **`ChatDTO`**: data transfer object di uscita, costruito dal controller a partire dagli oggetti `Chat`.
- **`Message`**: rappresenta l'entità di dominio che incapsula le informazioni relative a un singolo messaggio all'interno di una `Chat`.
- **`MessageDTO`**: data transfer object utilizzato per esporre i singoli messaggi all'esterno.
- **`ChatEntity`**: rappresenta la mappatura dell'entità «Chat» nel database PostgreSQL, gestita tramite *TypeORM*$_G$. E' associata a una tabella generata automaticamente e viene utilizzata per persistere ogni conversazione tra l'utente e BuddyBot. I suoi campi sono stati citati e spiegati nella sezione precedente durante la spiegazione della medesima classe.
- **`LastUpdateEntity`**: rappresenta l'entità incaricata di tracciare la data dell'ultimo *Retrieval periodico*$_G$ effettuato, ovvero l'ultimo momento in cui è stato eseguito un fetch globale delle informazioni. Nel database, la tabella *last_update* ospita un unico record persistente, contenente esclusivamente la data di aggiornamento più recente.

```
1  import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';
2
3  @Entity('last_update')
4  export class LastUpdateEntity {
5    @PrimaryGeneratedColumn()
6    id: number;
7
8    @Column({type: 'timestamp' })
9    lastFetch: Date;
10 }
```

5.5.4) Inserimento dell'ultima data di recupero informazioni

- **`LastUpdateDTO`**: data transfer object utilizzato per rappresentare il payload della richiesta in arrivo. Contiene un unico campo lastFetch, espresso come stringa, che rappresenta la data da registrare come ultimo fetch delle informazioni.
- **`InsertLastUpdateController`**: punto di ingresso del microservizio per la richiesta di aggiornamento del dato relativo all'ultimo retrieval. Il consumer (ossia tale controller) resta in ascolto di nuovi messaggi sulla coda "lastFetch_queue" ed espone un metodo insertLastRetrieval() che riceve come input un `LastUpdateDTO`, che trasformerà poi in un command object (Cmd). Ritornerà infine un boolean per rappresentare l'esito dell'operazione.
- **`LastUpdateCmd`**: si tratta del command object utilizzato per incapsulare e strutturare il dato passato dal DTO, prima di invocare lo *UseCase*. Questo passaggio consente di isolare il formato esterno (DTO) dalla logica interna, mantenendo un'interfaccia pulita verso il dominio applicativo.

- **`InsertLastUpdateUseCase`**: interfaccia che rappresenta la porta di ingresso della logica applicativa per l'inserimento in database di una nuova «data di ultimo aggiornamento». Utile per garantire disaccopiamento tra *Controller* e *Service*.
- **`InsertLastUpdateService`**: implementazione concreta dell'interfaccia precedente, è la classe principale della business logic. Non interagisce direttamente con il database, il suo ruolo è quello di **orchestrare** un corretto inserimento in database della data ottenuta. In linea con i principi dell'architettura esagonale, questa classe consente di mantenere la logica di business indipendente dall'infrastruttura.
- **`InsertLastUpdatePort`**: questa interfaccia rappresenta la porta di uscita (output port) dal punto di vista della logica applicativa. Astrae il meccanismo mediante il quale viene eseguito il processo di inserimento del dato in questione nel database.
- **`InsertLastUpdateAdapter`**: implementazione concreta dell'interfaccia spiegata in precedenza, funge da punto di collegamento tra logica applicativa al sistema di persistenza, richiamando il metodo richiesto ma non accedendo al database.
- **`ChatRepository`**: componente incaricata dell'accesso diretto a PostgreSQL, utilizzando *TypeORM*$_G$ per la gestione delle entità e delle query. In questo contesto, espone il metodo insertLastRetrieval(), che si occupa di aggiornare il valore della data di ultimo accesso nel record persistito della tabella last_update. Si individua il record con id = 1 (ossia unico record presente nella tabella) aggiornando il campo lastFetch con il dato nuovo da inserire.

  A seguire, viene inserito il metodo «insertLastRetrieval()» presente in questa classe.

```
...
export class ChatRepository {
  constructor(
    @InjectRepository(ChatEntity) //tabella db della chat
    private readonly chatRepo: Repository<ChatEntity>,

    @InjectRepository(LastUpdateEntity) //tabella db con unico record data ultimo retrieval info
    private readonly lastUpdateRepo: Repository<LastUpdateEntity>,
  ) {}


  async insertLastRetrieval(date: string): Promise<boolean> {
    const parsedDate = new Date(date);

    //id sempre 1
    const existing = await this.lastUpdateRepo.findOne({ where: { id: 1 } });

    if (existing) {
      existing.lastFetch = parsedDate;
      await this.lastUpdateRepo.save(existing);
    } else {
      const newEntry = this.lastUpdateRepo.create({
        id: 1,
        lastFetch: parsedDate,
      });
      await this.lastUpdateRepo.save(newEntry);
    }

    return true;
  }
  ...
}
```

- **`LastUpdateEntity`**: rappresenta l'entità incaricata di tracciare la data dell'ultimo *Retrieval periodico*$_G$ effettuato, ovvero l'ultimo momento in cui è stato eseguito un fetch globale delle informazioni. Nel database, la tabella *last_update* ospita un unico record persistente, contenente esclusivamente la data di aggiornamento più recente.

5.5.5) Ottenimento della data di ultimo recupero / aggiornamento informazioni

- **`LastUpdateDTO`**: data transfer object utilizzato per trasmettere verso l'esterno il valore corrente della data di ultimo aggiornamento.
- **`FetchLastUpdateController`**: rappresenta il punto di ingresso del microservizio per la richiesta di lettura della data relativa all'ultimo retrieval periodico effettuato dal sistema. Il consumer (ossia tale controller) resta in ascolto di nuovi messaggi sulla coda "getLastFetch_queue" ed espone un metodo fetchLastUpdate(). Una volta ricevuto un messaggio, attiva il metodo il quale delega l'elaborazione al caso d'uso implementato nel `FetchLastUpdateService`.
- **`FetchLastUpdateUseCase`**: interfaccia che rappresenta la porta di ingresso della logica applicativa per il recupero del dato dal database. Utile per garantire disaccopiamento tra *Controller* e *Service*.
- **`FetchLastUpdateService`**: implementazione concreta dell'interfaccia precedente, è la classe principale della business logic. Non interagisce direttamente con il database, il suo ruolo è quello di **orchestrare** un corretto recupero della «data di ultimo aggiornamento delle informazioni» presente in database. In linea con i principi dell'architettura esagonale, questa classe consente di mantenere la logica di business indipendente dall'infrastruttura.
- **`FetchLastUpdatePort`**: questa interfaccia rappresenta la porta di uscita (output port) dal punto di vista della logica applicativa. Astrae il meccanismo mediante il quale viene eseguito il processo di recupero del dato in questione dal database.
- **`FetchLastUpdateAdapter`**: implementazione concreta dell'interfaccia spiegata in precedenza, funge da punto di collegamento tra logica applicativa al sistema di persistenza, richiamando il metodo richiesto ma senza accedere al database.
- **`ChatRepository`**: componente incaricata dell'accesso diretto a PostgreSQL, utilizzando *TypeORM*$_G$ per la gestione delle entità e delle query. In questo contesto viene esposto il metodo fetchLastUpdate(), responsabile del recupero dell'unico record presente nella tabella *last_update*, contenente la data dell'ultimo retrieval periodico. Si individua il record con `id = 1` (ossia unico record presente nella tabella) e, una volta recuperato, viene restituito al chiamante.

  A seguire, viene inserito il metodo «fetchLastUpdate()» presente in questa classe.

```
async fetchLastUpdate(): Promise<LastUpdateEntity> {
  const entity = await this.lastUpdateRepo.findOne({ where: { id: 1 } });
  if (!entity) {
    throw new Error('LastUpdate-record not found (in db)');
  }
  return entity;
}
```

- **`LastUpdate`**: rappresenta l'entità di dominio, contiene un solo campo lastFetch, espresso come stringa, che identifica il momento in cui è stato eseguito l'ultimo fetch periodico delle informazioni.
- **`LastUpdateEntity`**: rappresenta l'entità incaricata di tracciare la data dell'ultimo *Retrieval periodico*$_G$ effettuato, ovvero l'ultimo momento in cui è stato eseguito un fetch globale delle informazioni. Nel database, la tabella *last_update* ospita un unico record persistente, contenente esclusivamente la data di aggiornamento più recente.

5.6) Microservizio Informazioni

Il microservizio Informazioni gestisce l'interazione tra il sistema e tre fonti esterne — **Jira**, **Confluence** e **GitHub** — recuperando informazioni rilevanti che vengono archiviate in un database vettoriale. Oltre alla fase di acquisizione e persistenza dei dati, il microservizio espone funzionalità di retrieval semantico, fornendo le informazioni più pertinenti in base alle query degli utenti per supportare il modello linguistico nella generazione di risposte contestualizzate.

5.6.1) Funzionalità principali

Il microservizio si articola in quattro casi d'uso fondamentali:
- Recupero e memorizzazione dei dati da **Jira**;
- Recupero e memorizzazione dei dati da **Confluence**;
- Recupero e memorizzazione dei dati da **GitHub**;
- Recupero di informazioni rilevanti basato sulle query utente.

Tutte le richieste vengono ricevute in modalità asincrona tramite **RabbitMQ**, che opera come message broker. Ogni messaggio attiva il caso d'uso corrispondente, gestito secondo un'architettura esagonale che garantisce una netta separazione tra logica di dominio, servizi applicativi e adattatori per l'integrazione con fonti esterne e sistema di storage.

5.6.2) Classi condivise

5.6.2.1) Qdrant-information-repository

Questa classe gestisce la persistenza e il retrieval delle informazioni nel database vettoriale Qdrant, fungendo da punto centralizzato per le operazioni di salvataggio e recupero. Il repository viene inizializzato con un'istanza del Vector Store di LangChain passata come attributo nel costruttore, permettendo un'astrazione efficace rispetto all'implementazione specifica del database vettoriale.

Operazioni principali:

- `storeInformation(info: Information): Result`

  Questo metodo gestisce il salvataggio di nuove informazioni nel database vettoriale attraverso i seguenti passaggi:
  1. Estrazione dei metadati dall'oggetto Information;
  2. Verifica dell'esistenza di vettori precedenti con lo stesso identificativo;
  3. Rimozione di eventuali vettori esistenti utilizzando i Metadata per garantire la consistenza;
  4. Suddivisione del documento in segmenti più piccoli (chunking) se la dimensione supera la soglia massima per un embedding efficace;
  5. Generazione degli embedding per ogni segmento attraverso il model provider gestito da LangChain;
  6. Salvataggio dei vettori risultanti nel database vettoriale attraverso l'uso Vector Store di LangChain.

  Il processo di chunking è particolarmente importante per gestire documenti di grandi dimensioni, assicurando che ogni segmento possa essere correttamente vettorializzato mantenendo al contempo la coerenza semantica.

- `retrieveRelevantInfo(query: string, k?: number): InformationEntity*`

  Implementa una ricerca semantica utilizzando i metodi nativi di LangChain per il Vector Store:
  1. Conversione della query testuale in un vettore embedding;
  2. Esecuzione di una ricerca di similarità nel database vettoriale;

3. Recupero dei `k` documenti più rilevanti (dove `k` è configurabile, con un valore predefinito);
4. Ordinamento dei risultati in base al punteggio di similarità;

La ricerca semantica permette di identificare documenti concettualmente simili alla query dell'utente, anche quando non condividono esattamente gli stessi termini, grazie alla rappresentazione vettoriale dello spazio semantico.

5.6.2.2) Metadata

Contiene informazioni supplementari relative agli oggetti di business salvati nel database vettoriale. Questi metadati identificano tutti i vettori derivati da un oggetto originale, consentendo modifiche o rimozioni precise dei dati nel database.

5.6.2.3) Information

Rappresenta un oggetto di business completo dei relativi **Metadata**, garantendo il corretto salvataggio del contenuto documentale. Questa classe assicura che gli oggetti provenienti da Jira, GitHub e Confluence e i loro metadati siano salvati coerentemente.

5.6.2.4) InformationEntity

Entità di repository per **Information**, agisce come DTO per la persistenza.

5.6.2.5) MetadataEntity

Entità di repository per **Metadata**, essenziale per l'identificazione e gestione dei vettori.

5.6.2.6) Result

Classe di supporto che fornisce un meccanismo standardizzato per rappresentare l'esito di operazioni di recupero e salvataggio dati. Permette di distinguere tra successo e fallimento, e in caso di errore, di fornire una descrizione dettagliata.

5.6.3) Recupero e memorizzazione dei dati da GitHub

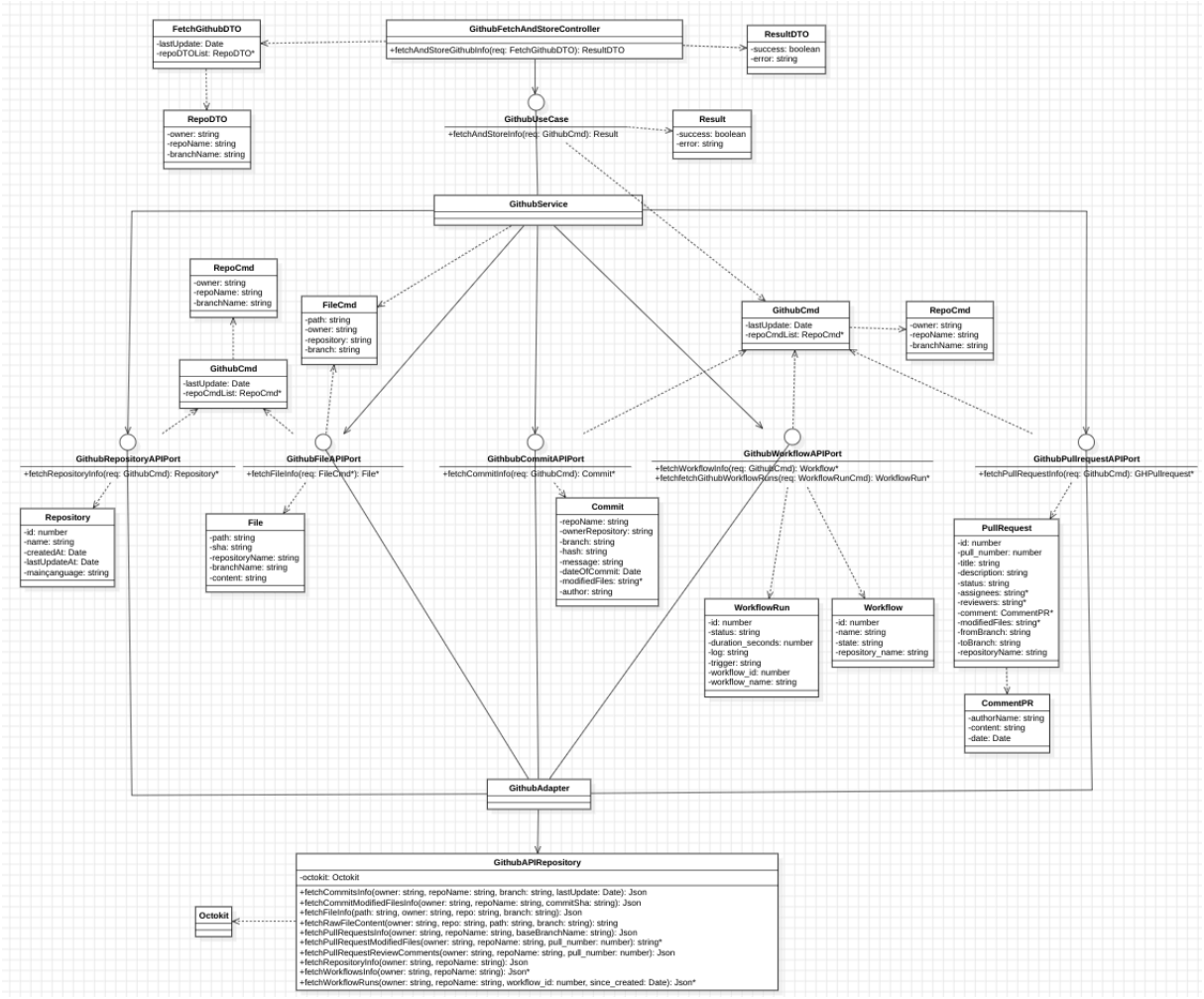

Figura 30: Diagramma UML di dettaglio riguardo alla raccolta delle informazioni di Github

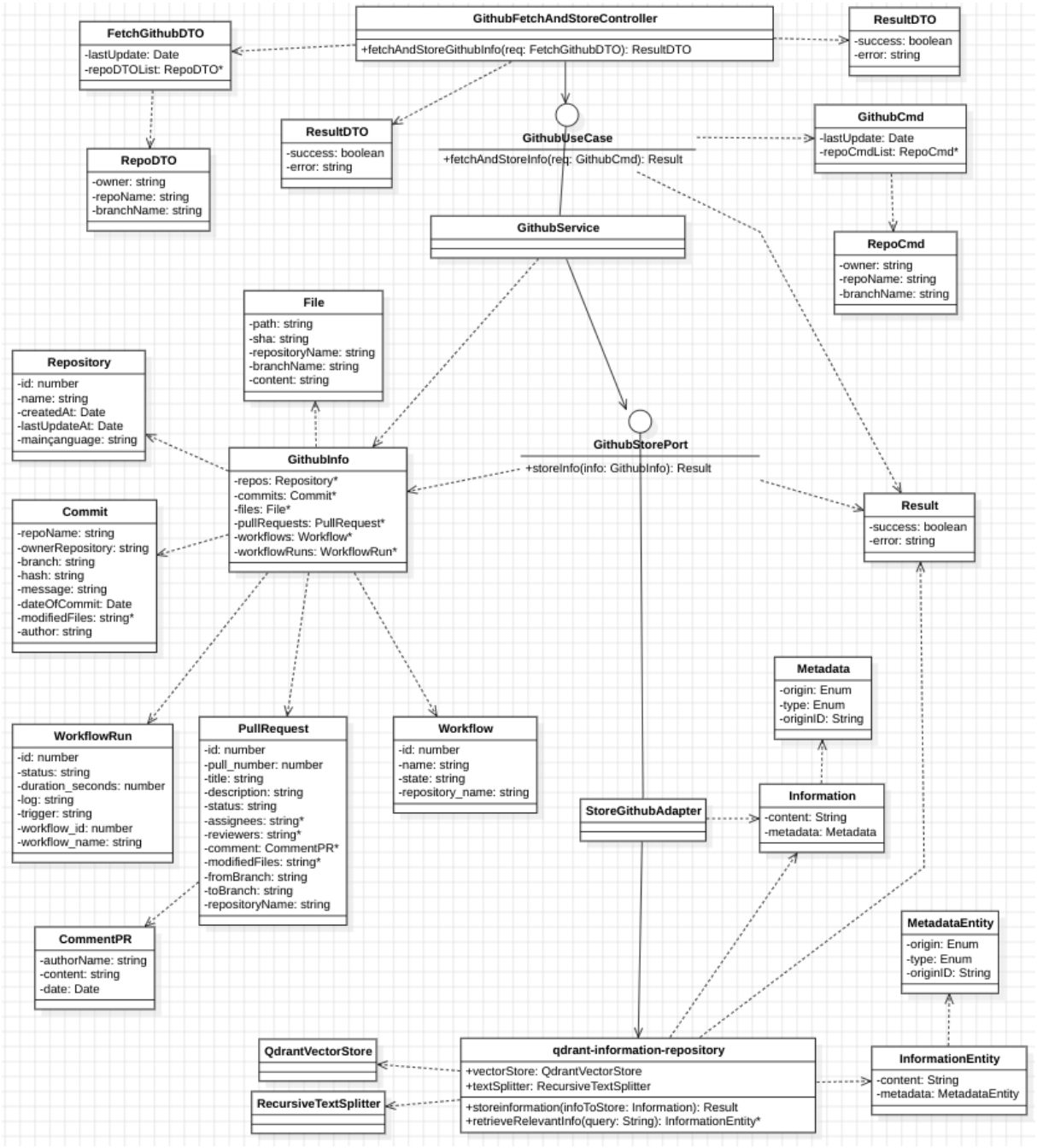

Figura 31: Diagramma UML di dettaglio riguardo al salvataggio delle informazioni di Github

5.6.3.1) FetchGithubDTO

Classe che viene ricevuta in input dall'`InformationController`, contiene una lista di `RepoDTO`, spiegati in seguito, e la data dall'ultima raccolta di informazioni.

```
export class FetchGithubDto {
  constructor (
    private repoDTOList: RepoGithubDTO[],
    private lastUpdate?: Date
  ){}
}
```

5.6.3.2) RepoDTO

Classe che contiene le informazioni necessarie a identificare univocamente la risorsa di cui vogliamo raccogliere le informazioni, ossia:
- a chi appartiene il repository su *Github*$_G$
- il nome del repository
- il branch del repository

```
export class RepoGithubDTO{
  constructor(
    private owner: string,
    private repoName: string,
    private branch_name: string
  ){}
}
```

5.6.3.3) GithubCmd

Questa classe rappresenta il Command che riceve la business logic, contiene una lista di `RepoCmd`, che contiene gli stessi campi di `RepoDTO`, e lo stesso "lastUpdate" ricevuto nel `FetchGithubDto`

```
export class GithubCmd {
  constructor(
    private repoCmdList:RepoCmd[],
    private lastUpdate?:Date
  ){}
}
```

5.6.3.4) RepoCmd

Questa classe è la classe `RepoDTO` adattata alla business logic.

```
export class RepoCmd{
  constructor (
    private owner: string,
    private repoName: string,
    private branch_name: string
  ){}
}
```

5.6.3.5) Commit

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere dei commit di una determinata repository.

```
export class Commit{
    constructor(
        private repoName: string,
        private ownerRepository: string,
        private branch: string,
        private hash: string,
        private message: string,
        private dateOfCommit: string,
        private modifiedFiles: string[],
        private author: string,
    ) {}

    toStringifiedJson(): string {
        return JSON.stringify(this);
    }
```

```
    getMetadata(): Metadata {
      return new Metadata(Origin.GITHUB, Type.COMMIT, this.hash);
    }
}
```

5.6.3.6) File

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere dei files di un determinato branch in una determinata repository.

```
export class File{
    constructor(
        private path: string,
        private sha: string,
        private repositoryName: string,
        private branchName: string,
        private content: string
    ) {}

    toStringifiedJson(): string {
        return JSON.stringify(this);
    }

    getMetadata(): Metadata {
      return new Metadata(Origin.GITHUB, Type.FILE, this.sha);
    }
}
```

5.6.3.7) PullRequest

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere delle pull requests in una determinata repository.

```
export class PullRequest{
    constructor(
        private id: number,
        private pull_number: number,
        private title: string,
        private description: string,
        private status: string,
        private assignees: string[],
        private reviewers: string[],
        private comments: CommentPR[],
        private modifiedFiles: string[],
        private fromBranch: string,
        private toBranch: string,
        private repository_name: string,
    ) {}

    toStringifiedJson(): string {
        return JSON.stringify(this);
    }

    getMetadata(): Metadata {
      return new Metadata(Origin.GITHUB, Type.PULLREQUEST, this.id.toString());
    }
}
```

5.6.3.8) CommentPR

Questa classe è oggetto della business logic, è contenuta all'interno di `PullRequest` in quanto si occupa di contenere al suo interno le informazioni riguardanti un determinato commento di review su una `PullRequest`.

```
export class CommentPR{
    constructor(
        private authorName: string,
        private content: string,
        private date: Date
    ){}

    getAuthorName(): string {
        return this.authorName;
    }

    getContent(): string {
        return this.content;
    }

    getDate(): Date {
        return this.date;
    }
}
```

5.6.3.9) Repository

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere di una determinata repository.

```
export class Repository {
    constructor(
    private id: number,
    private name: string,
    private createdAt: string,
    private lastUpdate: string,
    private mainLanguage: string,
  ) {}

  toStringifiedJson(): string {
    return JSON.stringify(this);
  }

  getMetadata(): Metadata {
    return new Metadata(Origin.GITHUB, Type.REPOSITORY, this.id.toString());
  }
}
```

5.6.3.10) Workflow

Questa classe è oggetto della business logic, contiene le informazioni che vogliamo raccogliere dei workflow in una determinata repository.

```
export class Workflow{
  constructor(
    private id: number,
    private name: string,
```

```
    private state: string,
    private repository_name: string,
  ) {}

  toStringifiedJson(): string {
    return JSON.stringify(this);
  }

  getMetadata(): Metadata {
    return new Metadata(Origin.GITHUB, Type.WORKFLOW, this.id.toString());
  }
}
```

5.6.3.11) WorkflowRun

Questa classe è oggetto della business logic, è contenuta all'interno di `Workflow` in quanto si occupa di contenere al suo interno le informazioni riguardanti una determinata run di un `Workflow`.

```
export class WorkflowRun {
  constructor(
    private readonly id: number,
    private readonly status: string,
    private readonly duration_seconds: number,
    private log: string,
    private trigger: string,
    private workflow_id: number,
    private workflow_name: string
  ) {}

  toStringifiedJson(): string {
    return JSON.stringify(this);
  }

   getMetadata(): Metadata {
    return new Metadata(Origin.GITHUB, Type.WORKFLOW_RUN, this.id.toString());
  }
}
```

5.6.3.12) GithubFetchAndStoreController

Controller che resta in attesa di messaggi sulla coda `information-queue`, al fine di portare a termine le operazioni di raccolta e salvataggio delle informazioni ottenute da Github. Ritorna come output un oggetto `ResultDTO`.

5.6.3.13) GithubUseCase

Interfaccia che si comporta da porta d'ingresso alla business logic, offre il metodo `fetchAndStoreInfo`, che prende in input il `GithubCmd` ricevuto dal controller.

```
export interface GithubUseCase {
    fetchAndStoreGithubInfo(req: GithubCmd): Promise<Result>;
}
```

5.6.3.14) GithubService

La classe principale della business logic, che implementa `GithubUseCase` citato precedentemente. Si occupa di recuperare tutte le informazioni descritte nell'*Analisi dei Requisiti*$_G$ e di salvarle nel database vettoriale.

5.6.3.15) GithubCommitAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchCommitInfo` che riceve in input `GithubCmd` e ritorna in output una lista di `Commit`.

```
export interface GithubCommitAPIPort {
    fetchGithubCommitsInfo(req: GithubCmd): Promise<Commit[]>
}
```

5.6.3.16) GithubFileAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchGithubFilesInfo` che riceve in input `FileCmd` e ritorna in output una lista di `Commit`.

```
export interface GithubFilesAPIPort {
    fetchGithubFilesInfo(req: FileCmd[]): Promise<File[]>
}
```

5.6.3.17) GithubPullRequestAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchGithubPullRequestsInfo` che riceve in input `GithubCmd` e ritorna in output una lista di `PullRequest`.

```
export interface GithubPullRequestsAPIPort {
    fetchGithubPullRequestsInfo(req: GithubCmd): Promise<PullRequest[]>
}
```

5.6.3.18) GithubRepositoryAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchGithubRepositoryInfo` che riceve in input `GithubCmd` e ritorna in output una lista di `Repository`.

```
export interface GithubRepositoryAPIPort {
    fetchGithubRepositoryInfo(req: GithubCmd): Promise<Repository[]>
}
```

5.6.3.19) GithubWorkflowAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre i metodi:

- `fetchGithubWorkflowInfo` che riceve in input `GithubCmd` e ritorna in output una lista di `Workflow`.
- `fetchGithubWorkflowRuns` che riceve in input `WorkflowRunCmd` e ritorna in output una lista di `WorkflowRun`.

```
export interface GithubWorkflowsAPIPort {
    fetchGithubWorkflowInfo(req: GithubCmd): Promise<Workflow[]>
    fetchGithubWorkflowRuns(req: WorkflowRunCmd): Promise<WorkflowRun[]>
}
```

5.6.3.20) GithubAPIAdapter

Questa classe implementa:

- `GithubCommitAPIPort`
- `GithubFileAPIPort`
- `GithubPullRequestAPIPort`
- `GithubRepositoryAPIPort`
- `GithubWorkflowAPIPort`

ponendosi come adapter tra la business logic e la classe che si occupa di fare le richieste API, ossia `GithubAPIRepository`. Trasforma infatti gli oggetti JSON "grezzi" ritornati da quest'ultima e li trasforma negli oggetti della business logic.

5.6.3.21) GithubAPIRepository

Questa è la classe che si occupa di interfacciarsi direttamente con le API di Github. Esegue richieste tramite il client offerto da `octo-kit` e ritorna JSON con i dati "grezzi".

5.6.3.22) GithubStoreInfoPort

Questa è l'interfaccia che funge da porta d'uscita (outbound port) al fine di salvare i `GithubInfo` nel database vettoriale, offre il metodo `storeGithubInfo` che riceve in input una lista di `GithubInfo`.

```
export interface GithubStoreInfoPort {
    storeGithubInfo(req: GithubInfo): Promise<boolean>
}
```

5.6.3.23) GithubStoreInfoAdapter

Questa classe implementa `GithubStoreInfoPort`, si occupa di trasformare i `GithubInfo` in `Information` per poter essere usati dal `qdrant-information-repository` ed essere salvati sul database vettoriale.

5.6.4) Recupero e memorizzazione dei dati da Confluence

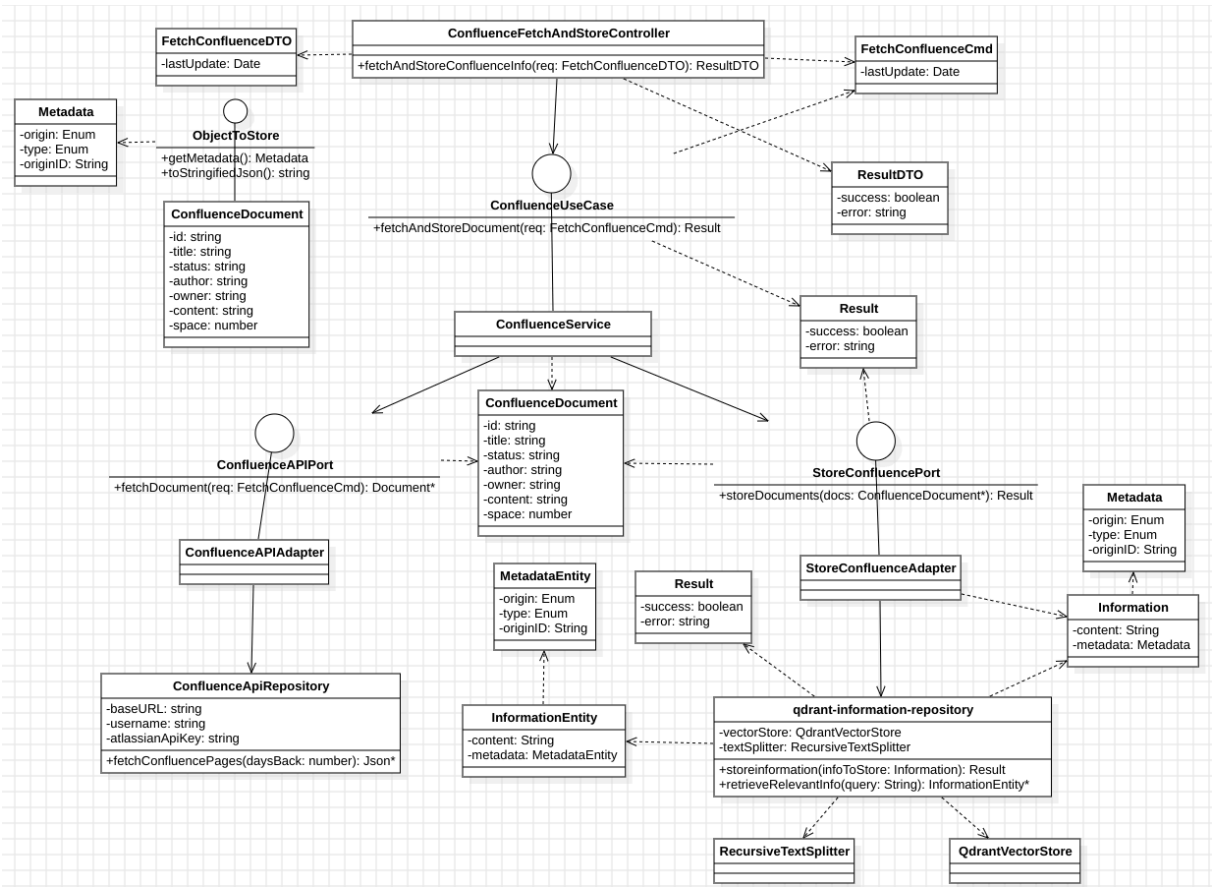

Figura 32: Diagramma UML di dettaglio riguardo a Confluence

5.6.4.1) ConfluenceController

Controller che resta in attesa di messaggi sulla coda `information-queue`, al fine di portare a termine le operazioni di raccolta e salvataggio delle informazioni ottenute da Confluence. Ritorna come output un oggetto `ResultDTO`.

5.6.4.2) ConfluenceUseCase

Interfaccia che si comporta da porta d'ingresso alla business logic, offre il metodo `fetchAndStoreDocument`, che prende in input il `ConfluenceCmd` ricevuto dal controller e ritorna come output un oggetto `Result`.

```
export interface ConfluenceUseCase {
  fetchAndStoreConfluenceInfo(req: ConfluenceCmd): Promise<Result>;
}
```

5.6.4.3) ConfluenceService

La classe principale della business logic, che implementa `ConfluenceUseCase` citato precedentemente. Si occupa di recuperare i documenti creati e modificati entro una certa data, presente all'interno di `ConfluenceCmd`.

```
export class ConfluenceService implements ConfluenceUseCase {
  constructor(
    @Inject(CONFLUENCE_API_PORT) private readonly confluenceAPIAdapter:
ConfluenceAPIPort,
    @Inject(CONFLUENCE_STORE_INFO_PORT) private readonly confluenceStoreAdapter:
ConfluenceStoreInfoPort
  ) {}

  async fetchAndStoreConfluenceInfo(req: ConfluenceCmd): Promise<Result> {
    const documents = await this.confluenceAPIAdapter.fetchDocuments(req);
    return await this.confluenceStoreAdapter.storeDocuments(documents);;
  }
}
```

5.6.4.4) ConfluenceDocument

Classe del domain, definisce le informazioni che vengono raccolte e viene usato come oggetto della business logic.

5.6.4.5) ConfluenceAPIPort

Interfaccia che si comporta come porta d'uscita (outbound port), offre il metodo `fetchDocuments` che riceve in input `ConfluenceCmd` e ritorna in output una lista di ConfluenceDocument.

5.6.4.6) ConfluenceAPIAdapter

Questa classe implementa `ConfluenceAPIPort`, ponendosi come adapter tra la business logic e la classe che si occupa di fare le richieste API, ossia `ConfluenceAPIRepository`. Trasforma infatti gli oggetti JSON "grezzi" ritornati da quest'ultima e li trasforma negli oggetti della business logic di `ConfluenceDocument`.

5.6.4.7) ConfluenceAPIRepository

Questa è la classe che si occupa di interfacciarsi direttamente con le API di Confluence. Esegue richieste HTTP e ritorna JSON con i dati "grezzi".

5.6.4.8) ConfluenceStorePort

Questa è l'interfaccia che funge da porta d'uscita (outbound port) al fine di salvare i `ConfluenceDocument` nel database vettoriale, offre il metodo `storeDocuments` che riceve in input una lista di `ConfluenceDocument`.

```
export interface ConfluenceStoreInfoPort {
  storeDocuments(req: ConfluenceDocument[]): Promise<Result>;
}
```

5.6.4.9) ConfluenceStoreAdapter

Questa classe implementa `ConfluenceStorePort`, si occupa di trasformare i `ConfluenceDocument` in `Information` per poter essere usati dal `qdrant-information-repository` ed essere salvati sul database vettoriale.

5.6.5) Recupero e memorizzazione dei dati da Jira

Il seguente diagramma illustra le classi coinvolte nel caso d'uso «Recupero e memorizzazione dei ticket di Jira», evidenziando l'architettura esagonale adottata:

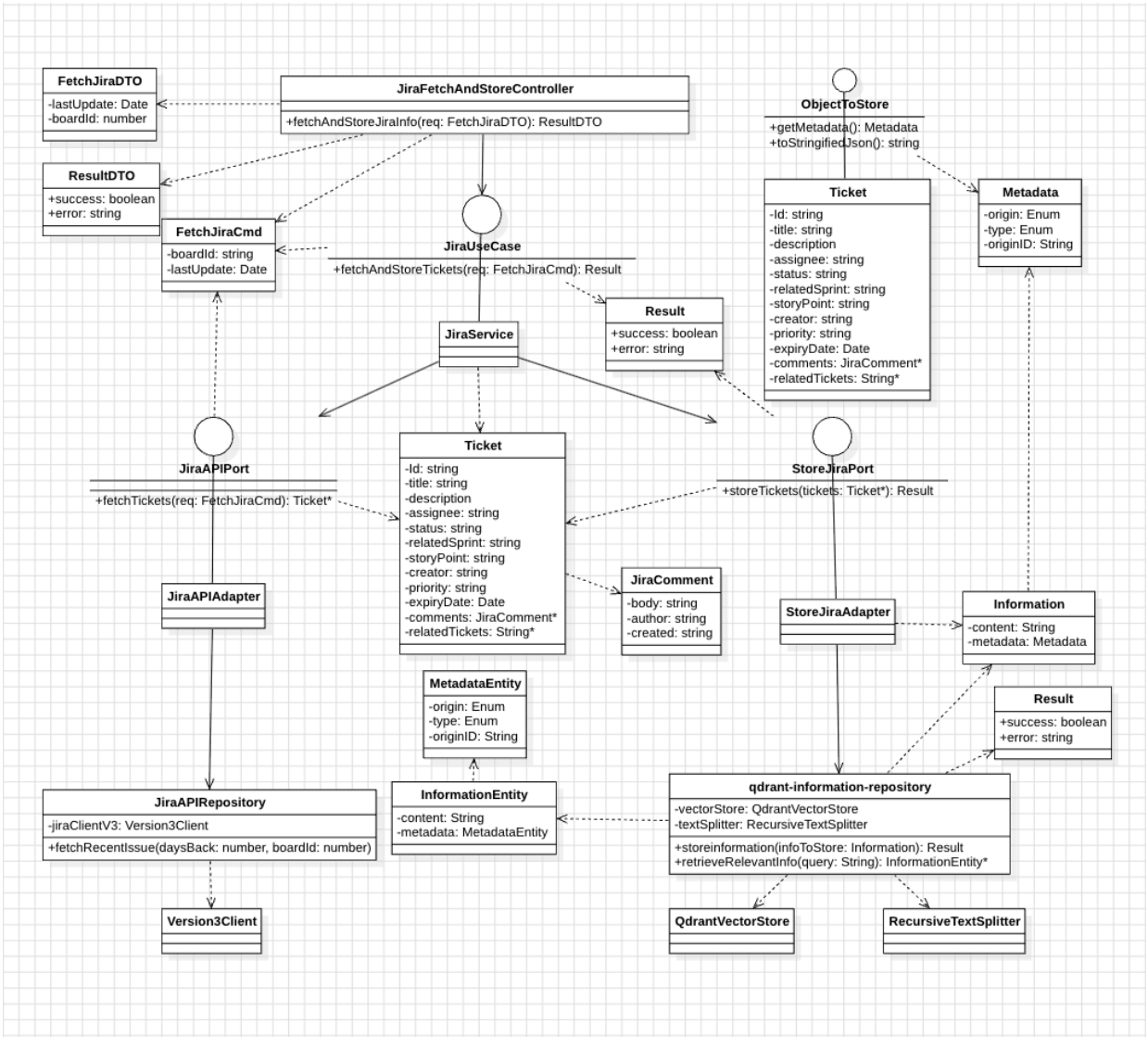

Figura 33: Diagramma delle classi per il caso d'uso di recupero e memorizzazione dei ticket di Jira

5.6.5.1) Componenti Principali

5.6.5.1.1) JiraFetchAndStoreController

Punto d'ingresso per l'operazione di recupero e memorizzazione dei ticket da Jira. Riceve le richieste esterne, le convalida e le indirizza verso il caso d'uso appropriato. Il controller accetta in input un `FetchJiraDTO` contenente tutte le informazioni necessarie, inclusa la data dell'ultimo aggiornamento per ottimizzare l'efficienza del recupero dati.

5.6.5.1.2) JiraUseCase

Interfaccia che definisce il contratto per la logica di recupero e memorizzazione, stabilendo una chiara astrazione tra definizione del comportamento e implementazione. Espone il metodo `WorkspaceAndStoreJiraInfo(req: FetchJiraCmd): Result` che restituisce un oggetto Result che rappresenta l'esito dell'operazione.

5.6.5.1.3) JiraService

Implementazione concreta di **JiraUseCase** che coordina:
1. Il recupero dei ticket tramite **JiraAPIPort**
2. L'elaborazione dei dati ottenuti
3. La memorizzazione mediante **StoreJiraPort**

Questo servizio incapsula la logica principale del caso d'uso, gestendo correttamente eventuali errori durante il processo.

5.6.5.1.4) Ticket

Rappresentazione strutturata di un ticket Jira nel dominio applicativo. Implementa l'interfaccia **ObjectToStore** fornendo implementazioni concrete dei metodi:
- `getMetadata(): Metadata`
- `toStringifiedJson(): string`

5.6.5.1.5) JiraComment

Modella i commenti associati a un ticket, includendo dettagli come autore, contenuto e timestamp, per una gestione completa delle informazioni correlate.

5.6.5.1.6) JiraAPIPort

Interfaccia che astrae le operazioni di interazione con l'API Jira, definendo un contratto chiaro indipendente dai dettagli implementativi. Espone il metodo `FetchTickets(req: FetchJiraCmd): Ticket*` che restituisce un array di ticket creati o modificati dalla data specificata nel comando.

5.6.5.1.7) JiraAPIAdapter

Implementa **JiraAPIPort** gestendo la comunicazione effettiva con l'API Jira tramite **JiraAPIRepository**. Traduce le risposte API nel formato interno richiesto dall'applicazione.

5.6.5.1.8) JiraAPIRepository

Classe che funge da intermediario per interagire direttamente con le API di Jira. Al momento della sua creazione, richiede l'iniezione di un client autenticato per stabilire la connessione con Jira. Espone un metodo `fetchRecentIssues(daysBack: number, boardId: number): Json*` in cui entrambi i parametri sono opzionali:
- **daysBack**: Specifica il numero di giorni nel passato per cui recuperare le issue. Se omesso, vengono restituite tutte le issue accessibili all'account;

- **boardId**: Limita la ricerca alle issue associate a una specifica board. Se omesso, vengono recuperate le issue da tutte le board accessibili.

5.6.5.1.9) StoreJiraPort

Definisce l'interfaccia per la memorizzazione dei ticket, permettendo al nucleo applicativo di salvare dati indipendentemente dal sistema di storage sottostante. Espone il metodo `storeTickets(tickets: Ticket*): Result` che restituisce l'esito dell'operazione.

5.6.5.1.10) StoreJiraAdapter

Implementa **StoreJiraPort** gestendo la persistenza dei ticket nel database vettoriale tramite **qdrant-information-repository**. Si occupa della trasformazione dei dati nel formato appropriato e dell'interazione con il meccanismo di storage.

5.6.6) Recupero di informazioni rilevanti basato sulle query utente

Il seguente diagramma illustra le classi coinvolte nel caso d'uso «Recupero delle informazioni rilevanti basato sulle query utente», evidenziando l'architettura esagonale adottata:

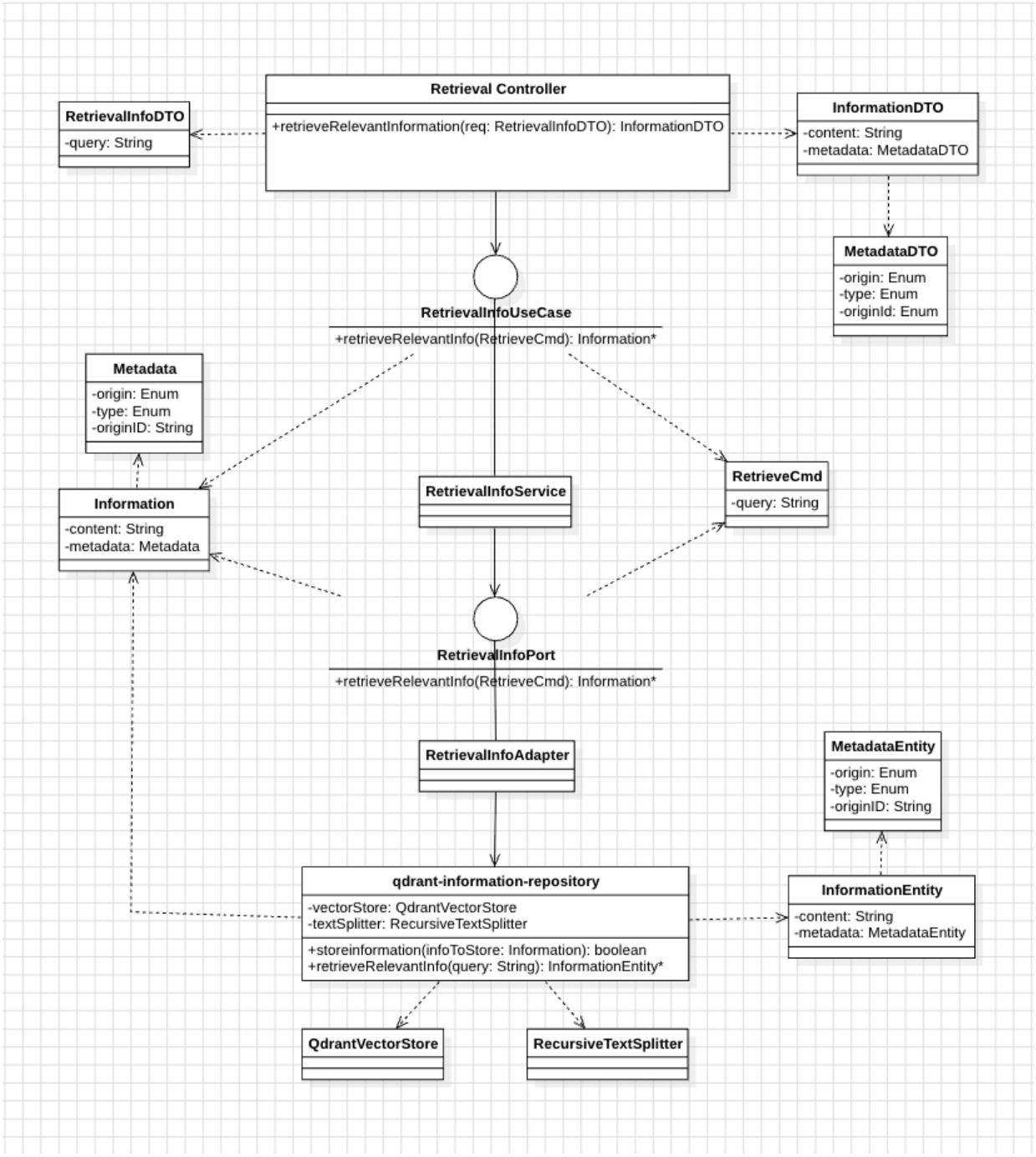

Figura 34: Diagramma delle classi per il caso d'uso di recupero di informazioni rilevanti basato sulle query utente

5.6.6.1) Componenti Principali

5.6.6.1.1) RetrievalController

Punto d'ingresso per il recupero delle informazioni. Riceve richieste esterne contenenti una stringa query incapsulata in un RetrievalInfoDTO, che viene poi convertito in un comando di dominio RetrieveCmd. Dopo aver invocato il caso d'uso, restituisce un array di oggetti InformationDTO che rappresentano le informazioni più rilevanti trovate.

5.6.6.1.2) RetrievalInfoUseCase

Interfaccia che definisce il contratto del caso d'uso, delegando la responsabilità di recuperare le informazioni rilevanti. Espone il metodo `retrieveRelevantInfo(cmd: RetrieveCmd): Information*`, che restituisce un array ordinato di oggetti Information.

5.6.6.1.3) RetrievalInfoService

Implementazione concreta di RetrievalInfoUseCase, si occupa della logica principale del caso d'uso. Riceve il comando, interagisce con la porta RetrievalInfoPort, la quale viene iniettata nel costruttore, e restituisce il risultato sotto forma di array di Information.

5.6.6.1.4) RetrieveCmd

Oggetto di dominio che incapsula la richiesta dell'utente, contenente il campo query. Utilizzato internamente per mantenere la coerenza del linguaggio di dominio tra i livelli.

5.6.6.1.5) RetrievalInfoPort

Interfaccia che astrae la logica di accesso ai dati. Espone il metodo `retrieveRelevantInfo(cmd: RetrieveCmd): Information*`, consentendo al servizio applicativo di restare disaccoppiato dalla tecnologia di persistenza.

5.6.6.1.6) RetrievalInfoAdapter

Implementazione concreta della porta RetrievalInfoPort, interagisce con qdrant-information-repository per recuperare le informazioni rilevanti alla domanda dell'utente. Converte gli oggetti provenienti dal repository in oggetti di dominio.

6) Tracciamento requisiti

6.1) Stato dei requisiti funzionali

Codice	Descrizione	Stato
RF-001	L'utente deve accedere all'applicazione senza necessità di autenticazione	Soddisfatto
RF-002	Il sistema deve archiviare in modo persistente le domande degli utenti e le risposte generate	Soddisfatto
RF-003	L'utente deve poter visualizzare lo storico della chat in ordine cronologico inverso (dal più recente al più vecchio).	Soddisfatto
RF-004	L'utente deve visualizzare un messaggio informativo che spiega che non ci sono messaggi nello storico	Soddisfatto
RF-005	L'utente deve visualizzare un messaggio di errore se il sistema non riesce a recuperare lo storico	Soddisfatto
RF-006	L'utente deve visualizzare un messaggio di errore se la richiesta non è stata completata a causa di un timeout	Soddisfatto
RF-007	L'utente deve visualizzare un messaggio di errore se il backend non è disponibile	Soddisfatto
RF-008	L'utente deve visualizzare per ogni messaggio: il contenuto, la data e ora di invio	Soddisfatto
RF-009	L'utente deve visualizzare lo sfondo di un messaggio inviato da un utente di colore grigio	Soddisfatto
RF-010	L'utente deve visualizzare lo sfondo di un messaggio inviato da ***Buddybot***$_G$ di colore blu	Soddisfatto
RF-011	L'utente deve visualizzare per ogni messaggio inviato da ***Buddybot***$_G$ la data e l'ora dell'ultimo aggiornamento dei dati usati per generare la risposta	Soddisfatto
RF-012	L'utente deve poter scrivere una domanda in linguaggio naturale	Soddisfatto
RF-013	L'utente deve poter inviare la domanda scritta al sistema	Soddisfatto
RF-014	L'utente deve poter visualizzare la risposta generata da ***Buddybot***$_G$	Soddisfatto
RF-015	L'utente deve essere informato se la domanda che ha posto non rientra nelle competenze specifiche del sistema tramite una risposta generata da ***Buddybot***$_G$	Soddisfatto
RF-016	L'utente deve essere informato se i documenti richiesti nella domanda non sono disponibili all'interno del sistema tramite una risposta generata da ***Buddybot***$_G$	Soddisfatto
RF-017	L'utente deve poter visualizzare un messaggio di errore se si è verificato un errore generico nella generazione della risposta da parte del ***backend***$_G$	Soddisfatto
RF-018	L'utente deve poter visualizzare un messaggio di errore se la risposta non è stata generata perchè supera la lunghezza massima consentita	Soddisfatto
RF-019	L'utente deve poter visualizzare un messaggio di errore se la domanda supera la lunghezza massima consentita	Soddisfatto

RF-020	Il sistema deve generare una risposta appropriata alla domanda posta dell'utente	Soddisfatto
RF-021	Il sistema deve recuperare da GitHub le seguenti informazioni: Per ogni repository: • Nome della repository • Id della repository • Descrizione della repository • Data di creazione della repository • Ultima data di aggiornamento della repository • Linguaggio principale della repository Per ogni commit: • Hash del commit • Messaggio del commit • Data e ora dell'ultimo commit • Branch associato al commit • File modificati nel commit • Autore dell'ultimo commit • Nome della repository di appartenenza del commit • Nome del branch di appartenenza del commit Per ogni pull request: • Id della pull request • Titolo della pull request • Descrizione della pull request • Stato della pull request • Assegnatario della pull request • Reviewers della pull request • Commenti della pull request • File modificati nella pull request • Branch di origine della pull request • Branch di destinazione della pull request • Nome repository di appartenenza Per ogni workflow: • Id del workflow • Nome del workflow • Stato del workflow • Nome repository di appartenenza • Lista delle run per il workflow Per ogni workflow run: • Id della run • Stato della run • Durata in secondi della run • Link del log della run • Trigger della run • Id del workflow di appartenenza • Nome del workflow di appartenenza	Soddisfatto

	Per ogni file: • Path del file • SHA del file • Nome repository di appartenenza • Nome branch di appartenenza • Contenuto del file	
RF-022	Il sistema deve recuperare da Confluence le seguenti informazioni: • Id di una pagina • Titolo di una pagina • Stato di una pagina • Autore di una pagina • Owner di una pagina • Spazio di una pagina • Contenuto di una pagina	Soddisfatto
RF-023	Il sistema deve recuperare da Jira le seguenti informazioni: • Id di un ticket • Titolo di un ticket • Descrizione di un ticket • Assegnatario di un ticket • Stato di un ticket • Sprint di appartenenza di un ticket • Story point estimate di un ticket • Creatore di un ticket • Priorità • Data di scadenza • Ticket collegati • Commenti del ticket Per ogni commento del ticket: • Autore del commento • Data di creazione • Contenuto del commento	Soddisfatto
RF-024	Il sistema deve informare l'utente in caso di errore durante la generazione della risposta	Soddisfatto
RF-025	Il sistema deve informare l'utente se la risposta supera la lunghezza massima consentita	Soddisfatto
RF-026	Il sistema deve fornire la data e l'ora dell'ultimo aggiornamento dei dati utilizzati	Soddisfatto
RF-027	Il sistema deve aggiornare i dati dei documenti provenienti da GitHub, Confluence e Jira ogni 24 ore	Soddisfatto
RF-028	Il sistema deve salvare i dati provenienti dalle fonti (Githbu, Jira, Confluence) in un database vettoriale	Soddisfatto
RF-029	Il sistema deve convertire i dati provenienti dalle fonti (Githbu, Jira, Confluence) da forma testuale a forma vettoriale	Soddisfatto
RF-030	L'utente deve poter modificare una domanda già inviata	Non soddisfatto
RF-031	L'utente deve poter selezionare il tema chiaro o scuro per visualizzare l'interfaccia utente	Soddisfatto

RF-032	Il sistema deve visualizzare un'icona identificativa (cliccabile ed interattiva) per l'accesso a una risorsa esterna, aprendo la pagina web associata in una nuova finestra o scheda del browser.	Soddisfatto
RF-033	Il sistema deve visualizzare un'icona identificativa (cliccabile ed interattiva) per l'accesso al sito-documentazione di Jira	Soddisfatto
RF-034	Il sistema deve visualizzare un'icona identificativa (cliccabile ed interattiva) per l'accesso al sito-documentazione di GitHub	Soddisfatto
RF-035	Il sistema deve visualizzare un'icona identificativa (cliccabile ed interattiva) per l'accesso al sito-documentazione di Confluence	Soddisfatto
RF-036	Il sistema deve visualizzare un'animazione di caricamento circolare durante il recupero dello storico della chat	Soddisfatto
RF-037	Il sistema deve visualizzare un'animazione di caricamento composta da tre puntini, durante l'elaborazione della risposta da parte del backend	Soddisfatto
RF-038	Il sistema deve visualizzare un pulsante «Load More» nella parte superiore della chat, che consenta all'utente di caricare 10 messaggi precedenti non ancora visualizzati	Soddisfatto
RF-039	L'utente deve visualizzare il contenuto del messaggio in formato markdown	Soddisfatto
RF-040	L'utente deve poteer incollare nell'input di testo il contenuto copiato in precedenza	Soddisfatto
RF-041	L'interfaccia utente deve scrollare verso il basso mostrando l'ultimo messaggio inviato ogni volta che l'utente invia un nuovo messaggio	Soddisfatto

Tabella 1: Stato Requisiti Funzionali

6.2) Grafici riassuntivi

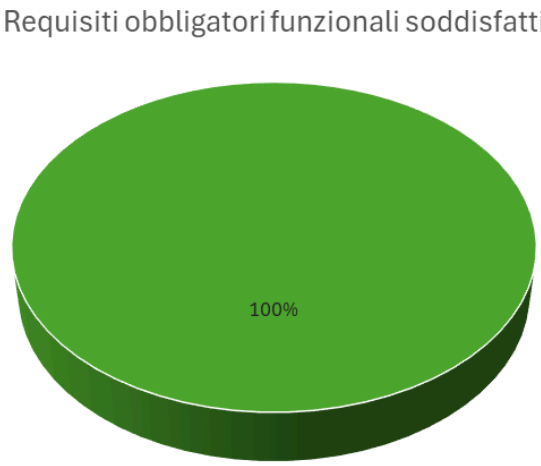

Figura 35: Stato dei requisiti funzionali obbligatori

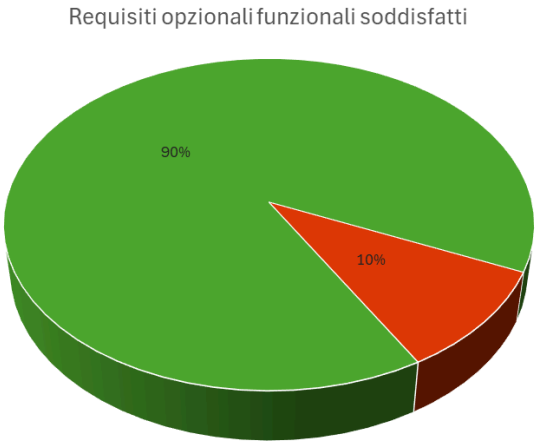

Figura 36: Stato dei requisiti funzionali opzionali

Figura 37: Stato dei requisiti funzionali desiderabili