Università degli Studi di Padova
Laurea: Informatica
Corso: Ingegneria del Software
Anno Accademico: 2024/2025

Gruppo: SWEg Labs
Email: gruppo.sweg@gmail.com

Specifica Tecnica

Versione 1.0.0

Stato	Approvato
Redazione	Federica Bolognini Michael Fantinato Giacomo Loat Filippo Righetto Riccardo Stefani Davide Verzotto
Verifica	Federica Bolognini Michael Fantinato Giacomo Loat Filippo Righetto Riccardo Stefani Davide Verzotto
Proprietario	Giacomo Loat
Uso	Esterno
Destinatari	Prof. Tullio Vardanega Prof. Riccardo Cardin *AzzurroDigitale Srl*

Registro delle modifiche

Versione	Data	Descrizione	Autore	Verificatore
1.0.0	25-03-25	Corretto il link al Glossario nella sezione §**1.5**	Giacomo Loat	Riccardo Stefani
0.3.0	24-03-25	Aggiornamento della catena di GetMessages nelle sezioni §**3.4.8** e §**4** dopo l'implementazione dell'infinite scroll	Riccardo Stefani	Filippo Righetto
0.2.8	21-03-25	Inseriti screenshot dei diagrammi delle classi e completata sezione §**3.4**	Riccardo Stefani	Giacomo Loat
0.2.7	18-03-25	Terminata scrittura sezione §**5**	Riccardo Stefani	Giacomo Loat
0.2.6	18-03-25	Iniziata scrittura sezione §**5**	Filippo Righetto	Riccardo Stefani
0.2.5	17-03-25	Scritte le sezioni §**3.4.3**, §**3.4.4**, §**3.4.5** e §**4.2**	Riccardo Stefani	Michael Fantinato
0.2.4	17-03-25	Scritta la sezione §**3.1**	Michael Fantinato	Riccardo Stefani
0.2.3	09-03-25	Scritte le sezioni §**3.4.6** e §**3.4.9**, e aggiunta la descrizione di tutte le relative classi nella sezione §**4**	Riccardo Stefani	Giacomo Loat
0.2.2	09-03-25	Scritte le sezioni §**3.4.7** e §**3.4.8**, e aggiunta la descrizione di tutte le relative classi nella sezione §**4**	Giacomo Loat	Riccardo Stefani
0.2.1	08-03-25	Scritte le sezioni §**3.4.2** e §**3.5**	Riccardo Stefani	Giacomo Loat
0.2.0	08-03-25	Corretti alcuni simboli del glossario inseriti sbagliati	Riccardo Stefani	Giacomo Loat
0.1.11	07-03-25	Scritta la sezione §**3.4.1**	Riccardo Stefani	Michael Fantinato
0.1.10	05-03-25	Inizializzazione della struttura delle sezioni §**3.4**, §**3.5** e §**4**	Riccardo Stefani	Giacomo Loat
0.1.9	25-02-25	Aggiunta sezione §**3.3** riguardante l'architettura di deployment	Filippo Righetto	Riccardo Stefani
0.1.8	24-02-25	Aggiunta sezione §**3** e stesura architettura logica	Davide Verzotto	Riccardo Stefani
0.1.7	13-02-25	Aggiunto PostgreSQL alla sezione §**2.1.3**	Giacomo Loat	Riccardo Stefani
0.1.6	11-02-25	Sistemati i link presenti nella sezione §**1.5** seguendo i consigli del professor Vardanega	Riccardo Stefani	Giacomo Loat
0.1.5	10-02-25	Aggiunti PyTest e Jasmine alla sezione §**2.2.2**	Riccardo Stefani	Giacomo Loat
0.1.4	10-02-25	Aggiunto FastAPI alla sezione §**2.1.1**, Angular e Node Docker alla sezione §**2.1.2** e GPT-4o alla sezione §**2.1.4**	Michael Fantinato	Riccardo Stefani
0.1.3	10-02-25	Aggiunti Python e LangChain alla sezione §**2.1.1** e aggiunto Docker alla sezione §**2.1.4**	Giacomo Loat	Riccardo Stefani

Versione	Data	Descrizione	Autore	Verificatore
0.1.2	08-02-25	Scrittura sezione §**1**	Federica Bolognini	Riccardo Stefani
0.1.1	06-02-25	Scrittura sezioni §**2.1.3** e §**2.2.1**	Riccardo Stefani	Giacomo Loat
0.1.0	06-02-25	Creazione del documento	Riccardo Stefani	Federica Bolognini

Tabella 1: Registro delle modifiche

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Scopo del documento

Questo documento fornisce linee guida per gli sviluppatori incaricati dell'estensione o della manutenzione del prodotto.
Al suo interno sono raccolte tutte le informazioni sui linguaggi e le tecnologie adottate, sull'architettura del sistema e sulle scelte progettuali effettuate.

1.2 Scopo del prodotto

Nell'ultimo anno vi è stato un cambiamento repentino nello sviluppo e nell'applicazione dell'*Intelligenza Artificiale*$_G$ all'elaborazione e raccomandazione dei contenuti alla generazione di essi, come immagini, testi e tracce audio. Il *capitolato*$_G$ C9, "BuddyBot", pone come obiettivo la realizzazione di un applicativo che permetta di porre interrogazioni in linguaggio naturale sullo stato attuale dei progetti software in lavorazione, ricevendo una risposta il quanto più precisa. Tale risposta dovrà essere generata tramite un *LLM*$_G$ collegato. Tale software sarà fruibile attraverso un'*applicazione web*$_G$, dove l'utente potrà interrogare il chatbot sullo stato attuale del codice e della documentazione dei progetti software nelle piattaforme utilizzate per il loro sviluppo.

1.3 Glossario

Al fine di prevenire ed evitare possibili ambiguità nei termini e acronimi presenti all'interno della documentazione, è stato realizzato un glossario nel file *glossario_v2.0.0.pdf* in grado di dare una definizione precisa per ogni vocabolo potenzialmente ambiguo. All'interno di ogni documento i termini specifici, che quindi hanno una definizione all'interno del *Glossario*$_G$, saranno contrassegnati con una $_G$ aggiunta a pedice e scritti in corsivo. Tale prassi sarà rispettata solamente per la prima occorrenza del termine in una determinata sezione del documento.

1.4 Miglioramenti al documento

La revisione e l'evoluzione del documento sono aspetti fondamentali per garantirne la *qualità*$_G$ e l'adeguatezza nel tempo. Questo consente di apportare modifiche in base alle esigenze concordate tra i membri del gruppo e il *proponente*$_G$. Di conseguenza, questa versione del documento non può essere considerata definitiva o completa, in quanto soggetta a possibili aggiornamenti futuri.

1.5 Riferimenti

1.5.1 Riferimenti normativi

- **Norme di Progetto v.2.0.0**;
- **Capitolato d'appalto C9 - BuddyBot (slide 3-18)**:
  https://www.math.unipd.it/ tullio/IS-1/2024/Progetto/C9.pdf
  (Ultimo accesso: 03/04/2025);
- **Regolamento progetto didattico (slide 2-25)**:
  https://www.math.unipd.it/ tullio/IS-1/2024/Dispense/PD1.pdf
  (Ultimo accesso: 03/04/2025).

1.5.2 Riferimenti informativi

- **Glossario v.2.0.0**.

2 Tecnologie coinvolte

Questa sezione fornisce un'analisi esaustiva delle tecnologie impiegate nel progetto in questione, comprendendo gli strumenti e le librerie necessarie per lo sviluppo, il testing e la distribuzione del prodotto. Saranno discusse le tecnologie utilizzate per implementare il *backend*$_G$, il *frontend*$_G$, la gestione dei dati, l'integrazione con i servizi esterni previsti, e l'analisi del codice, statica e dinamica.

2.1 Tecnologie utilizzate per la codifica

2.1.1 Strumenti per il backend

Nome	Versione	Descrizione
Python	3.13.1	Linguaggio di programmazione ad alto livello, interpretato, orientato agli oggetti e multiparadigma.
LangChain	0.3.18	LangChain è un *framework*$_G$ open-source progettato per sviluppare applicazioni che sfruttano i *Large Language Models*$_G$.
FastAPI	0.115.6	FastAPI è un *framework* web moderno e performante per lo sviluppo di *API*$_G$ in *Python*$_G$. Sfrutta le funzionalità avanzate di *Python* per la dichiarazione dei tipi, la validazione automatica dei dati e la generazione di documentazione interattiva, semplificando lo sviluppo e il mantenimento di applicazioni scalabili.

Tabella 2: Strumenti per il backend

2.1.2 Strumenti per il frontend

Nome	Versione	Descrizione
Angular	19.0.6	Angular è un *framework*$_G$ open source sviluppato da Google per la realizzazione di applicazioni web a pagina singola. Basato su *Typescript*$_G$, offre un'architettura modulare e funzionalità avanzate per la creazione di interfacce utente dinamiche e scalabili.
Node Docker	22-alpine	Il container *Docker*$_G$ utilizzato si basa sull'immagine ufficiale di *Node.js*$_G$ 22-alpine. Questa immagine, costruita su *Alpine Linux*$_G$, offre un ambiente di esecuzione leggero e performante, ideale per applicazioni in produzione, grazie alla sua ridotta impronta e all'ottimizzazione delle risorse.

Tabella 3: Strumenti per il frontend

2.1.3 Strumenti di gestione dei dati

Nome	Versione	Descrizione
Chroma	0.6.4	Chroma è un *database vettoriale*$_G$ progettato per memorizzare e gestire vettori ad alta dimensionalità. È ottimizzato per operazioni di ricerca e recupero efficienti, dunque aiuta a ridurre il numero di documenti di contesto da inviare all'*LLM*$_G$ selezionando solo quelli più rilevanti per la query corrente grazie alla ricerca di *similarità*$_G$ tra quest'ultima e i documenti salvati come vettori.
PostgreSQL	17.4	*PostgreSQL*$_G$, o Postgres, è un potente sistema di gestione di database relazionali ad oggetti open-source. È noto per la sua solidità, affidabilità e ricchezza di funzionalità. Supporta estensioni avanzate, transazioni complesse, e integrazioni con vari linguaggi di programmazione.

Tabella 4: Strumenti di gestione dei dati

2.1.4 Strumenti di integrazione e di supporto

Nome	Versione	Descrizione
GPT-4o	/	GPT-4o è un modello di linguaggio avanzato sviluppato da *OpenAI*$_G$, capace di comprendere ed elaborare testi complessi scritti in linguaggio naturale. Grazie alla sua architettura basata su deep learning, offre elevata coerenza, contestualizzazione e creatività nella generazione di contenuti.
Docker	27.3.1	Sistema di containerizzazione che permette la distribuzione di software in ambienti isolabili, dove ogni contenitore viene gestito in modo indipendente rispetto agli altri.

Tabella 5: Strumenti di integrazione e di supporto

2.2 Strumenti per l'analisi del codice

2.2.1 Strumenti per l'analisi statica

Nome	Versione	Descrizione
Pylint	3.3.4	Pylint è un analizzatore di codice statico e permette di analizzare codice *Python*$_G$ senza eseguirlo. Controlla la presenza di errori, applica uno standard di codifica e cerca di dare suggerimenti su come il codice potrebbe essere modificato.
SonarQube for IDE	4.15.1	Si tratta di un'estensione per *IDE*$_G$ che aiuta a rilevare e correggere i problemi di qualità durante la scrittura del codice, individuando i difetti in modo che possano essere corretti prima del commit del codice. Non è specifico per linguaggio, bensì supporta tanti linguaggi differenti.
ESLint	9.16.0	ESLint è uno strumento di analisi del codice statico per identificare e segnalare pattern trovati nel codice *JavaScript*$_G$ e *TypeScript*$_G$, cioè uno dei linguaggi utilizzati da *Angular*$_G$. Aiuta a mantenere uno stile di codifica coerente e a prevenire errori comuni, fornendo suggerimenti per migliorare la qualità del codice.

Tabella 6: Strumenti per l'analisi statica

2.2.2 Strumenti per l'analisi dinamica

Nome	Versione	Descrizione
PyTest	8.3.4	PyTest è un framework di testing per *Python*$_G$ che consente di scrivere test semplici e scalabili. Supporta test unitari, funzionali e di integrazione, offrendo funzionalità avanzate come fixture, parametri e plugin per estendere le sue capacità.
Jasmine	5.4.0	Jasmine è un framework di testing per *JavaScript*$_G$ che permette di scrivere test unitari in modo semplice e leggibile. Fornisce un'ampia gamma di funzionalità per la scrittura di test asincroni, la gestione delle aspettative e la creazione di suite di test modulari, aiutando a garantire la qualità e la robustezza del codice. È particolarmente utile per testare applicazioni sviluppate con *Angular*$_G$, grazie alla sua integrazione con il framework.

Tabella 7: Strumenti per l'analisi dinamica

3 Architettura

Questa sezione fornisce una descrizione dettagliata dell'*architettura*$_G$ del prodotto software, illustrando le scelte progettuali adottate per garantire la corretta realizzazione del sistema. Saranno presentate le principali scelte, *pattern architetturali*$_G$, e i *componenti software*$_G$ che compongono il sistema.

3.1 Logica del Prodotto

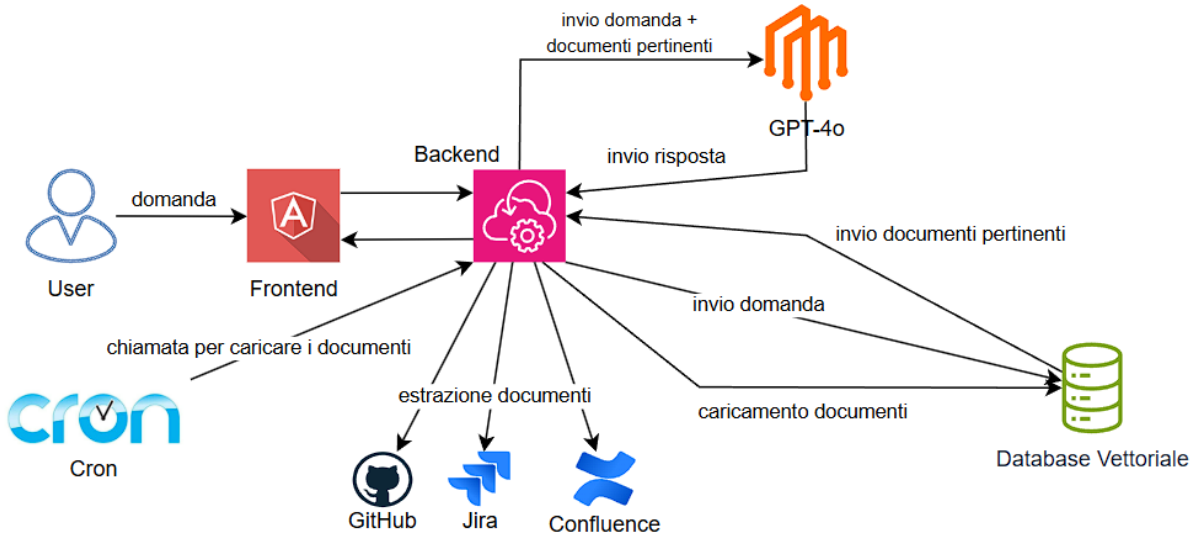

Figura 1: Logica del Prodotto

L'applicazione web BuddyBot permette di porre domande inerenti all'ecosistema aziendale sfruttando un *Large Language Model (LLM)*$_G$, nella fattispecie GPT-4o, integrato con un *database vettoriale*$_G$, cioè un sistema di archiviazione basato su tecniche di embedding. La qualità delle risposte è garantita dall'integrazione diretta con documenti e dati provenienti dalle principali piattaforme interne (*GitHub*$_G$, *Jira*$_G$ e *Confluence*$_G$), sincronizzati automaticamente ogni 24 ore da un *cron*$_G$. Tutti i documenti recuperati vengono suddivisi in parti più piccole, dette *chunk*$_G$, e trasformati in vettori numerici. Questi vettori, insieme ai relativi metadati, vengono archiviati stabilmente nel *database vettoriale Chroma*$_G$, consentendo di effettuare rapidamente ricerche semantiche mirate. Quando l'utente invia una domanda attraverso l'interfaccia web, il *frontend*$_G$ inoltra la richiesta al *backend*$_G$. Quest'ultimo esegue una ricerca di *similarità*$_G$ all'interno del *database vettoriale*$_G$ per identificare i documenti più rilevanti in relazione alla domanda posta. Il *backend* unisce quindi la domanda originale dell'utente, i risultati della ricerca di similarità e un header, contenente istruzioni precise per il modello sul contesto aziendale di riferimento e sul formato desiderato della risposta, generando un unico prompt che viene inviato al modello. Il modello genera dunque una risposta contestualizzata, integrando automaticamente una sezione denominata *"Link correlati"*, contenente tutti i riferimenti ai documenti aziendali utilizzati nella formulazione della risposta stessa. Qualora la domanda dell'utente risulti chiaramente fuori contesto aziendale, il chatbot risponderà con *"La domanda è fuori contesto"*. Nel caso in cui non siano presenti documenti o informazioni pertinenti nel *database vettoriale*$_G$, la risposta fornita sarà *"Informazione non trovata"*. Tutti i messaggi tra utente e chatbot vengono salvati in modo persistente nel *database relazionale*$_G$ *PostgreSQL*$_G$. Al primo caricamento dell'applicazione, l'interfaccia mostra automaticamente gli ultimi 50 messaggi scambiati. Se l'utente desidera consultare i messaggi precedenti, può scorrere verso l'alto, e ulteriori messaggi vengono caricati progressivamente suddivisi in pagine da 50 messaggi ciascuna. Inoltre, dopo ogni risposta, il *frontend* chiede al *backend*, che a sua volta chiede al modello GPT-4o, di generare 3 domande correlate all'ultima domanda e all'ultima risposta, che vengono offerte all'utente, stimolando ulteriori interazioni coerenti con la conversazione in corso. L'interfaccia permette infine di copiare rapidamente le risposte del chatbot, e anche di copiare singolarmente eventuali porzioni di codice presenti al loro interno. Per quanto riguarda la gestione degli errori, l'applicazione avvisa l'utente tramite messaggi dedicati qualora si riscontrassero problemi nel recupero dei messaggi dal database *PostgreSQL* o durante la sincronizzazione con le piattaforme aziendali. Anche eventuali difficoltà nella generazione di risposte o nella generazione delle domande suggerite vengono gestite informando tempestivamente

l'utente e consentendo, ove possibile, di continuare ad utilizzare il servizio in modalità limitata e/o di riprovare successivamente.

3.2 Architettura logica

L'*architettura logica*$_G$ adottata nella realizzazione dell'applicativo si basa sul modello di *architettura esagonale*$_G$. Il livello di *business*$_G$ del software è quindi indipendente dagli altri componenti, ovvero nulla presente all'esterno della logica di business può conoscere la sua *implementazione*$_G$. Questo principio alla base dell'architettura esagonale ci permette di ottenere un prodotto software facilmente testabile e *manutenibile*$_G$. Le componenti che devono restare indipendenti sono rappresentate dal *Domain business model*$_G$, che non comunica mai direttamente con l'esterno. Gli elementi che permettono il funzionamento dell'architettura esagonale sono i seguenti:

- ***Controller***$_G$: contiene l'*application logic*$_G$ del sistema: gestisce le richieste in ingresso, valida i dati di tipo *DTO*$_G$ ricevuti in input, e li adatta verso un tipo di dato di business. Il *Controller* ha poi il compito di chiamare uno *UseCase* per eseguire la logica di business, adattare l'output ricevuto in un oggetto DTO e restituirlo al client;
- ***UseCase***$_G$: rappresenta un caso d'uso specifico del sistema, che viene implementato da un *Service* per realizzare la logica di business;
- ***Service***$_G$: contiene la *business logic*$_G$ del sistema: esegue operazioni specifiche esclusive del dominio di interesse e delega le interazioni con le altre componenti ai *Port*. I *Service* possono interagire unicamente con tipi di dato di business, garantendo l'indipendenza della logica di business dal resto del sistema;
- ***Port***$_G$: definisce le interfacce attraverso le quali i *Service* interagiscono con il mondo esterno, permettendo il salvataggio e recupero di dati persistenti senza modificare la logica di business;
- ***Adapter***$_G$: implementa una o più interfacce definite dai *Port*, permettendo la comunicazione tra la logica di business e le tecnologie esterne di *persistent logic*$_G$. L'*adapter* si occupa di adattare i dati ricevuti dalla logica di business in un tipo *Entity*$_G$ adatto per la persistenza, e viceversa;
- ***Repository***$_G$: contiene la *persistent logic* del sistema: gestisce la persistenza dei dati, interagendo con un database o altre forme di storage per salvare e/o recuperare le informazioni necessarie. I tipi di dato gestiti dal *repository* sono gli *Entity*, che rappresentano il livello intermedio tra i tipi di business e i dati persistenti del sistema.

È stato scelto di utilizzare un'architettura esagonale per i seguenti motivi:

- **Facilità di test**: l'architettura esagonale permette di testare facilmente la business logic, in quanto è possibile sostituire gli adapter con degli *stub*$_G$ o dei *mock*$_G$;
- ***Manutenibilità***$_G$: l'architettura esagonale permette di mantenere la business logic indipendente dagli altri componenti, facilitando la manutenzione del codice;
- ***Scalabilità***$_G$: l'architettura esagonale permette di aggiungere nuove fonti di dati esterne semplicemente introducendo nuovi adapter, senza dover modificare la business logic.

3.3 Architettura di Deployment

Per determinare l'*architettura di deployment*$_G$ più adatta all'applicativo, si è tenuto conto del contesto reale in cui verrà utilizzato. Poiché il sistema è destinato all'utilizzo in un'azienda, dove non si prevedono significative espansioni o modifiche strutturali dopo l'installazione, la scelta di un'architettura monolitica è risultata la più appropriata. Questa soluzione, oltre a essere perfettamente in linea con le esigenze del prodotto, semplifica le fasi di progettazione, sviluppo e test. Inoltre, rispetto a un'architettura a microservizi, la monolitica evita complessità che sarebbero state difficili da gestire, anche considerando le competenze attuali del team di sviluppo.

Il deployment del prodotto viene gestito attraverso la containerizzazione con ***Docker Compose***$_G$. Questa scelta consente di semplificare l'installazione dell'applicativo, fornendo un ambiente preconfigurato in cui tutte le dipendenze sono già risolte. In questo modo, vengono predisposti automaticamente tutti i servizi necessari per garantire il corretto funzionamento del sistema, riducendo le difficoltà legate alla configurazione manuale.

3.4 Architettura di dettaglio

3.4.1 Architettura della generazione di una risposta

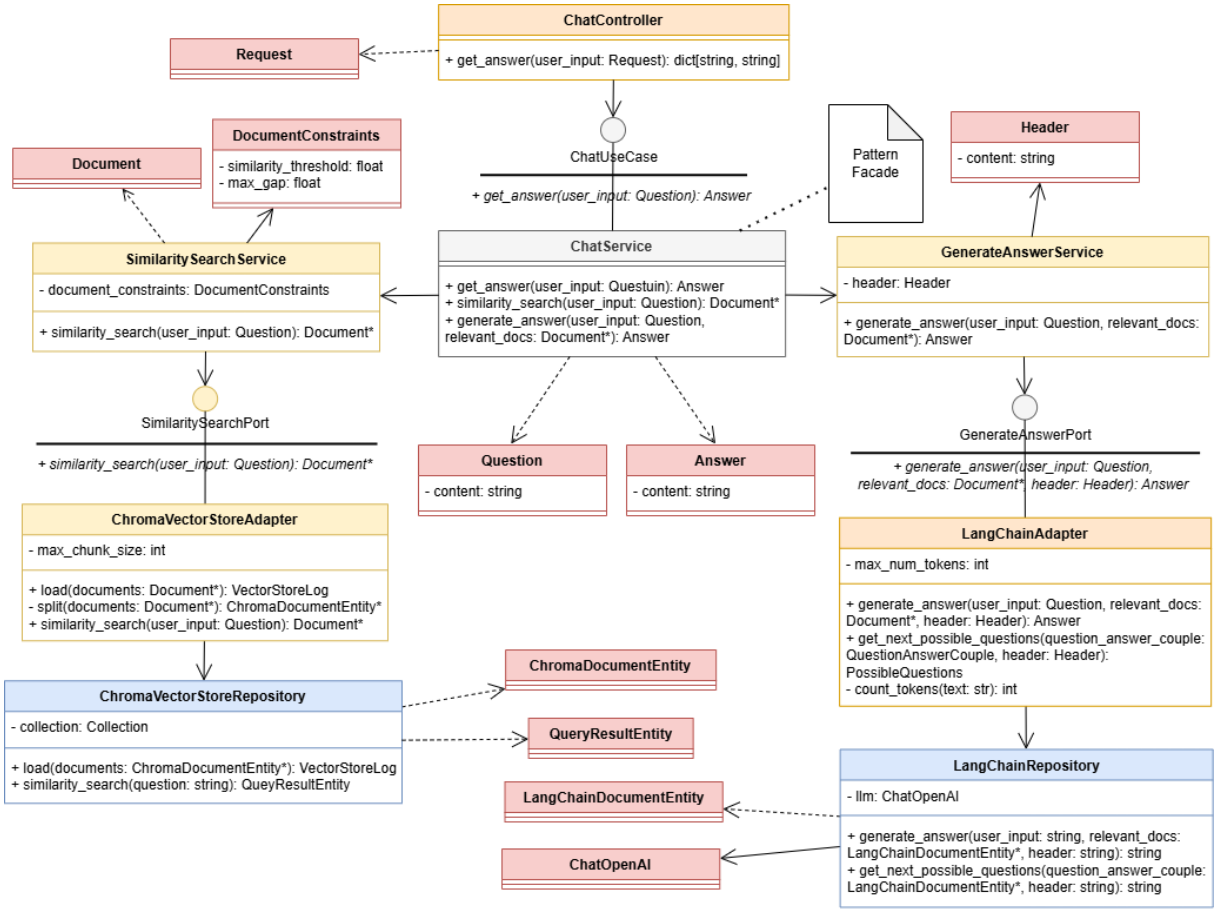

Figura 2: Architettura della generazione di una risposta

L'*UML*$_G$ in questione rappresenta l'*architettura*$_G$ della generazione di una risposta all'interrogazione di un utente, dal punto di vista del *backend*$_G$. Il *frontend*$_G$ invoca il metodo `chat` esposto dal *backend*$_G$ tramite *FastAPI*$_G$, passando la domanda dell'utente in forma di dizionario. La domanda viene recuperata dal ChatController, che converte la stringa interna al dizionario in un oggetto di Business, Question, e lo invia all'interfaccia ChatUseCase, implementata da ChatService. ChatService, che implementa il *Design Pattern Facade*$_G$, chiama SimilaritySearchService, componente responsabile della ricerca di *similarità*$_G$ nel *database vettoriale*$_G$, il quale si collega con SimilaritySearchPort verso l'esterno, sfruttando l'*architettura esagonale*$_G$. La Porta viene implementata da ChromaVectorStoreAdapter, che adatta la Question verso un tipo stringa inviato a ChromaVectorStoreRepository, che gestisce il collegamento con il database vettoriale *Chroma*$_G$, sul quale avviene la ricerca di similarità. Una volta ottenuti i risultati della ricerca e trasformati in *Entity*$_G$, essi vengono convertiti nel tipo di Business Document e poi restituiti a ChatService, il quale dunque li invia assieme all'input dell'utente a GenerateAnswerService, classe di servizio munita di un attributo Header che consente di fornire un'introduzione di contesto al chatbot. Header, input utente e documenti rilevanti vengono quindi forniti a LangChainAdapter, che si assicura che la somma dei *token*$_G$ dei parametri non superi la soglia limite per l'*LLM*$_G$ configurato. Infine, i parametri vengono forniti a LangChainRepository, che li combina assieme in unico prompt e, sfruttando gli strumenti forniti dalla libreria *LangChain*$_G$, effettua una chiamata verso l'LLM di *OpenAI*$_G$ configurato per ottenere una risposta. La risposta viene quindi convertita in un oggetto Answer di *Business Logic*$_G$, ed infine convertita in stringa per essere inserita in un dizionario in ChatController, il quale restituisce la risposta al frontend.

3.4.2 Architettura dell'aggiornamento automatico del database vettoriale

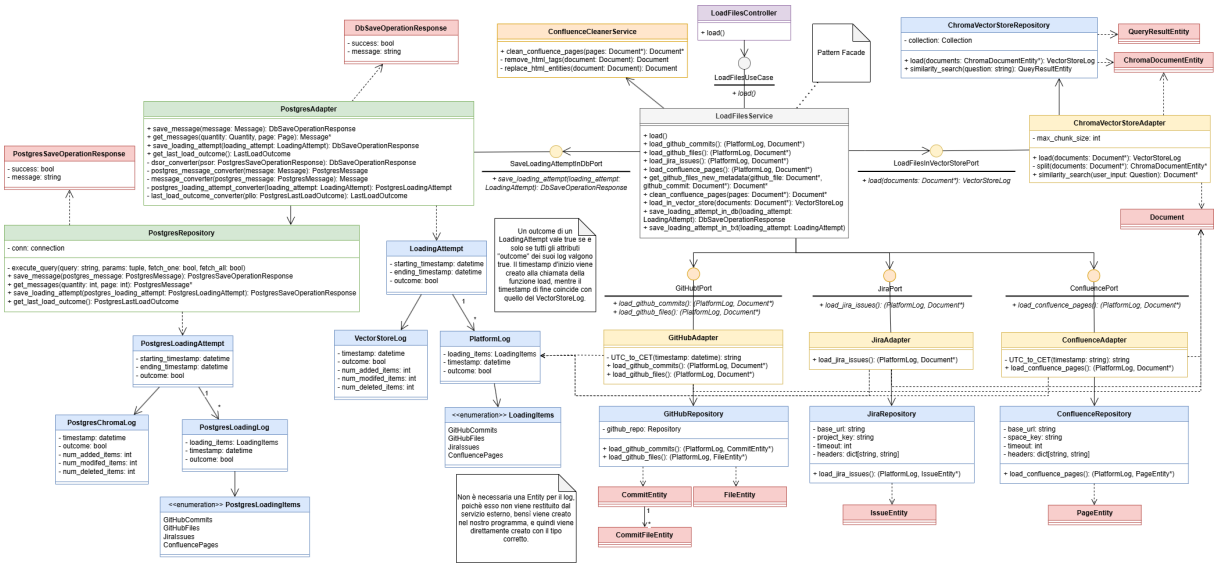

Figura 3: Architettura dell'aggiornamento automatico del database vettoriale

L'UML in questione rappresenta l'architettura dell'aggiornamento automatico del *database vettoriale*$_G$, il quale viene svolto da un *cron*$_G$ configurato periodicamente per eseguire un apposito script *Python*$_G$. Il cron chiama il metodo `load` della classe LoadFilesController, che a sua volta chiama l'interfaccia LoadFilesUseCase implementata da LoadFilesService.
LoadFilesService implementa il pattern *Facade*$_G$ per fornire un'unica interfaccia di collegamento alle piattaforme dalle quali caricare i file e alle piattaforme per il salvataggio degli stessi file e dei log di tutte le operazioni. In particolare, LoadFilesService si collega alle interfacce GitHubPort, JiraPort e ConfluencePort, che poi verranno implementate dai rispettivi adapter, che possiederanno i rispettivi repositories, rispettando l'*Architettura Esagonale*$_G$. Vengono quindi recuperati i Commits e i Files da *GitHub*$_G$, le Issues da *Jira*$_G$ e le Pagine da *Confluence*$_G$, estratti dagli account configurati, che vengono prima convertiti in oggetti *Entity* nei repositories, e poi nel tipo di business Document mediante gli adapter. I documenti corrispondenti ai file di GitHub, una volta tornati a LoadFilesService, vengono confrontati con i commit GitHub per rilevare la loro data di creazione, mentre i documenti corrispondenti alle pagine di Confluence, una volta tornati a LoadFilesService, vengono "puliti" da tag ed entities HTML grazie all'attributo di tipo ConfluenceCleanerService di quest'ultima classe. La lista di oggetti Document viene dunque inviata a LoadFilesInVectorStorePort, implementata da ChromaVectorStoreAdapter, che si occupa di adattare i Document in DocumentEntity per poi inviarli a ChromaVectorStoreRepository, che si occupa di salvare i documenti nel database vettoriale *Chroma*$_G$. Nel salvataggio, viene anche chiaramento distinto il numero di documenti aggiunti per la prima volta, il numero di documenti modificati ed il numero di documenti eliminati: viene considerata la data di ultimo aggiornamento per le Jira Issues, le Confluence Pages e i GitHub Commits, mentre avviene il confronto tra la data di creazione e la data di ultimo inserimento in Chroma per i GitHub Files.
Ogni classe di repository, oltre a svolgere il proprio di compito, si occupa anche di creare e restituire uno specifico oggetto di log, di tipo PlatformLog per GitHub, Jira e Confluence e di tipo VectorStoreLog per Chroma, che vengono uniti assieme in un unico oggetto LoadingAttempt e poi inviati all'interfaccia SaveLoadingAttemptInDbPort. Essa viene implementata da PostgresAdapter, che si occupa di adattare l'oggetto verso il tipo PostgresLoadingAttempt e di inviarlo a PostgresRepository, che lo salva nel *database relazionale*$_G$ *Postgres*$_G$, per tenere traccia dell'esito di ogni tentativo di caricamento di documenti nel database vettoriale. Successivamente, lo stesso log, opportunamente formattato, viene salvato in un file di testo, situato dentro la cartella `src/backend`.

3.4.3 Architettura del rendering dello storico, al caricamento iniziale e allo scroll

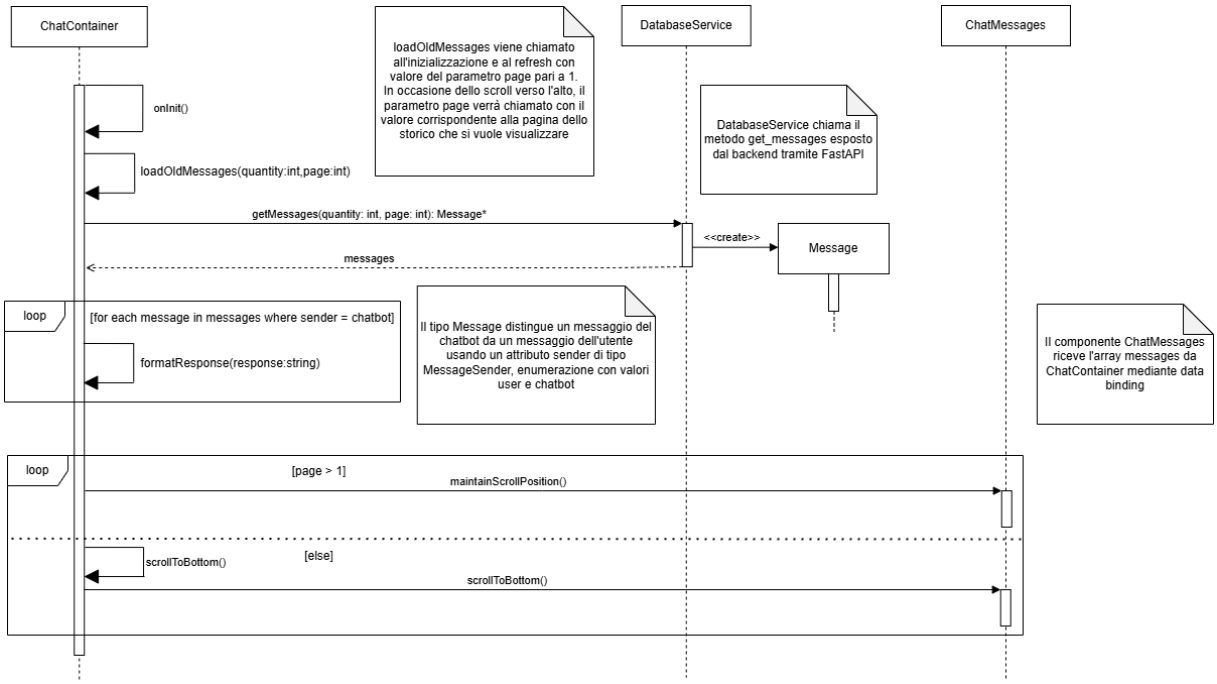

Figura 4: Architettura del rendering dello storico, al caricamento iniziale e allo scroll

L'UML in questione rappresenta l'architettura del rendering dello storico, al caricamento iniziale e allo scroll, cioè descrive tutte quelle operazioni compiute dai componenti e servizi di *Angular*$_G$ per garantire che, all'avvio ed allo scroll verso il basso, sia visualizzabile una lista di messaggi, recuperati dal backend, nella schermata principale della pagina web.
Il componente ChatContainer, alla sua istanziazione, chiama il proprio metodo `loadOldMessages`, che chiama `getMessages` di DatabaseService passando come parametri la quantità di messaggi da ottenere e il numero della pagina da cui prelevarli. Una pagina raggruppa un certo numero di messaggi caricati in unico passaggio: nel caso del caricamento iniziale, la pagina da caricare è la pagina 1, cioè la pagina dei messaggi più recenti, mentre, nel caso dello scroll, viene caricata la pagina successiva rispetto a quella che si sta visualizzando in quel momento (se si sta visualizzando la pagina 1, viene caricata la 2, se si sta visualizzando la pagina 4, viene caricata la 5, ecc.). Il metodo `getMessages` di DatabaseService invoca dunque il metodo `get_messages` esposto dal backend tramite FastAPI, passando come parametro un dizionario che indica la quantità di messaggi da ottenere e il numero di pagina, che lancia la catena descritta in §**3.4.8**. Una volta ottenuta la lista di oggetti serializzati dal backend, DatabaseService si occupa di creare una lista di oggetti di tipo Message, che restituisce a ChatContainer. ChatContainer, poi, si occupa di formattare appositamente, tra la lista dei messaggi, i messaggi scritti dal chatbot, tramite la funzione `formatResponse`, per evidenziare parole e zone di interesse delle risposte.
Poi, nel caso di caricamento iniziale, ChatContainer chiama il proprio metodo scrollToBottom, che chiama il metodo scrollToBottom di ChatMessages, che fa in modo che l'utente visualizzi nella schermata iniziale i messaggi più recenti. Altrimenti, nel caso di scroll verso il basso, e quindi di caricamento di messaggi sovrastanti quelli già presenti, è necessario ricalibrare la barra di scorrimento laterale senza che cambi la schermata visualizzata dall'utente, e dunque ChatContainer chiama l'apposita funzione maintainScrollPosition di ChatMessages.

3.4.4 Architettura del rendering grafico di domanda e risposta

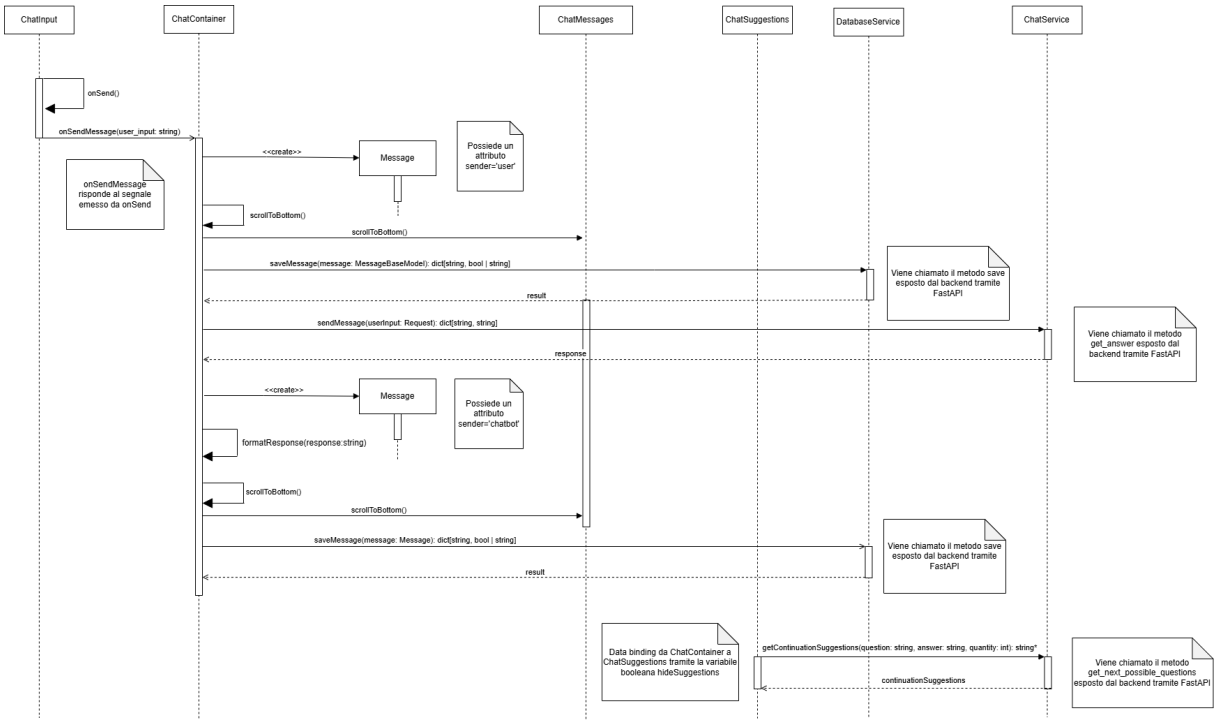

Figura 5: Architettura del rendering grafico di domanda e risposta

L'UML in questione rappresenta l'architettura del rendering grafico di domanda e risposta, cioè descrive tutte quelle operazioni compiute dai componenti e servizi di Angular durante l'interazione tra utente e chatbot per renderizzare visivamente la chat.
ChatInput, il componente che contiene la barra di input e il pulsante di invio, all'evento di invio di una domanda da parte dell'utente risponde con il metodo `onSend`, il quale chiama `onSendMessage` di ChatContainer, che inizialmente crea un oggetto Message e lo stampa visivamente a schermo, e poi chiama `scrollToBottom` per scorrere in basso. Poi, ChatContainer si occupa di far partire il salvataggio del messaggio chiamando il metodo `saveMessage` di DatabaseService, che a sua volta chiama il metodo `save_message` esposto dal backend tramite FastAPI, che segue la catena di classi descritta in §**3.4.7** fino a salvare il messaggio nel database relazionale Postgres. La risposta restituita con l'esito del salvataggio giunge a ChatContainer e viene stampata sulla console. ChatContainer chiama allora il metodo `sendMessage` di ChatService, che a sua volta chiama il metodo `get_answer` esposto da FastAPI che fa partire la catena descritta in §**3.4.1**. Una volta ottenuta la risposta in forma di stringa, ne viene creato il relativo oggetto di tipo Message e il suo contenuto viene formattato per evidenziare parole e zone di interesse, tramite la funzione `formatResponse`. ChatContainer chiama poi `scrollToBottom` per scorrere in basso e far visualizzare la risposta all'utente, ed il metodo `saveMessage` di DatabaseService per salvare il messaggio, ma, soprattutto, aggiorna l'attributo hideSuggestions a false, al quale è sottoscritto in *data binding*$_G$ il componente ChatSuggestions. Questo componente va dunque a chiamare il metodo `getContinuationSuggestions` di ChatService, che a sua volta chiama il metodo `get_continuation_suggestions` esposto dal backend tramite FastAPI, che segue la catena di classi descritta in §**3.4.9** per ottenere una lista di domande per proseguire la conversazione, generate anch'esse dall'LLM. Le domande vengono dunque visualizzate a schermo sopra la barra di input: al loro click, viene emesso un segnale, che viene ascoltato da ChatContainer, che, non appena lo riceve, chiama, come sopra, il metodo `onSendMessage`, facendo sì che la selezione di una domanda suggerita da parte dell'utente sia equivalente alla scrittura e invio di una domanda libera.

3.4.5 Architettura frontend dell'aggiornamento del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico

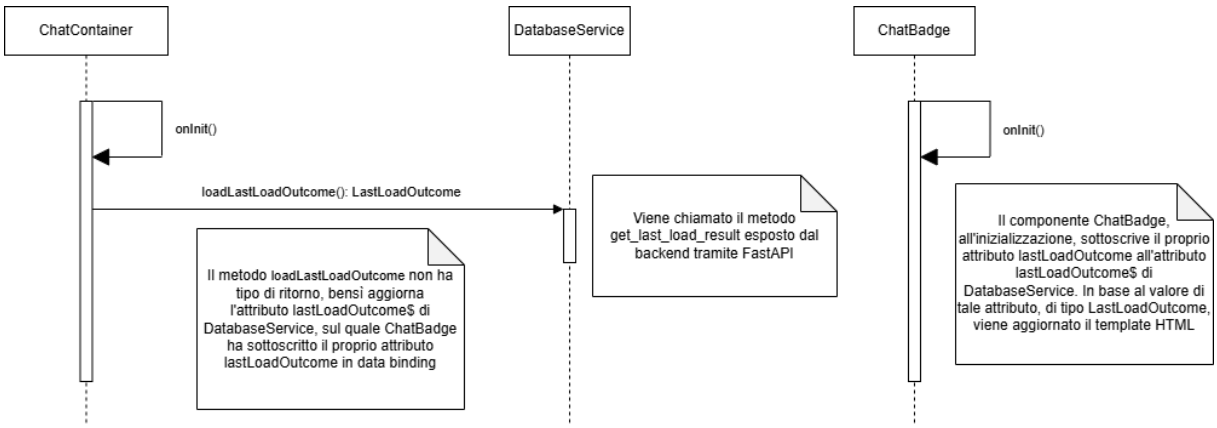

Figura 6: Architettura frontend dell'aggiornamento del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico

L'UML in questione rappresenta l'architettura del rendering grafico del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico, cioè descrive tutte quelle operazioni compiute dai componenti e servizi di Angular per renderizzare visivamente il badge.
Il componente ChatBadge, alla sua istanziazione, sottoscrive il proprio attributo lastLoadOutcome in *data binding* all'attributo lastLoadOutcome$ di DatabaseService. Il componente ChatContainer, alla sua istanziazione, esegue il metodo `loadLastLoadOutcome` di DatabaseService, il quale chiama il metodo `get_last_load_outcome` esposto dal backend tramite FastAPI, che segue la catena di classi descritta in §**3.4.6** per ottenere l'esito dell'ultimo aggiornamento automatico. DatabaseService dunque assegna il proprio attributo lastLoadOutcomeSubject al valore dell'esito, opportunamente convertito in un oggetto LastLoadOutcome, e tale valore viene osservato prima dall'altro attributo lastLoadOutcome$, e quindi poi da ChatBadge, che aggiorna il proprio attributo lastLoadOutcome. ChatBadge, infine, si occupa di formattare appositamente il badge in base al proprio attributo, in modo che sia visivamente chiaro se l'ultimo aggiornamento automatico è andato a buon fine o meno, oppure se non è stato possibile recuperare il suo esito.

3.4.6 Architettura backend dell'aggiornamento del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico

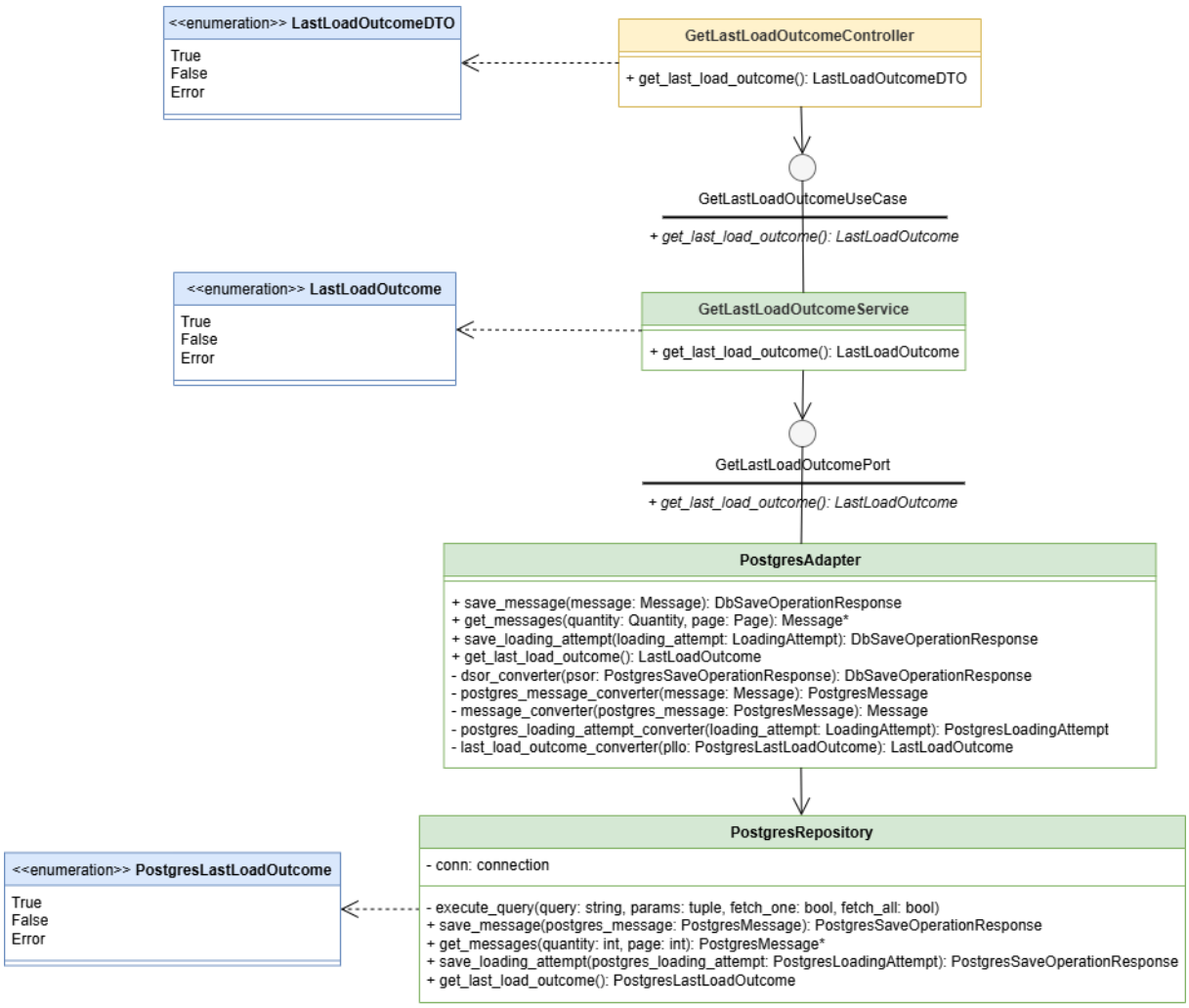

Figura 7: Architettura backend dell'aggiornamento del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico

L'UML in figura rappresenta l'architettura dell'aggiornamento del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico, dal punto di vista del backend. Il frontend invoca il metodo `get_last_load_outcome` esposto dal backend tramite FastAPI, il quale chiama GetLastLoadOutcomeController, che a sua volta chiama l'interfaccia GetLastLoadOutcomeUseCase implementata da GetLastLoadOutcomeService. GetLastLoadOutcomeService chiama l'interfaccia GetLastLoadOutcomePort, implementata da PostgresAdapter, che invoca il metodo `get_last_load_outcome` esposto da PostgresRepository, che si occupa di recuperare da Postgres l'ultimo esito del caricamento di documenti nel database vettoriale Chroma. Più precisamente, viene recuperato l'attributo `outcome` della tupla con timestamp di fine aggiornamento più recente dalla tabella `loading_attempts`.
PostgresRepository restituisce un oggetto di tipo PostgresLastLoadOutcome, che può assumere valore True, se l'ultimo tentativo di aggiornamento ha avuto esito positivo, False, se l'ultimo tentativo di aggiornamento ha avuto esito negativo, o Error, se c'è stato un errore nell'interazione con Postgres per recuperare l'esito dell'ultimo aggiornamento. L'oggetto di tipo PostgresLastLoadOutcome viene convertito in un oggetto di tipo LastLoadOutcome da PostgresAdapter e restituito a GetLastLoadOutcomeService. Infine, GetLastLoadOutcomeController converte l'oggetto verso il tipo LastLoadOutcomeDTO e lo restituisce al frontend.

3.4.7 Architettura del salvataggio dei messaggi nello storico

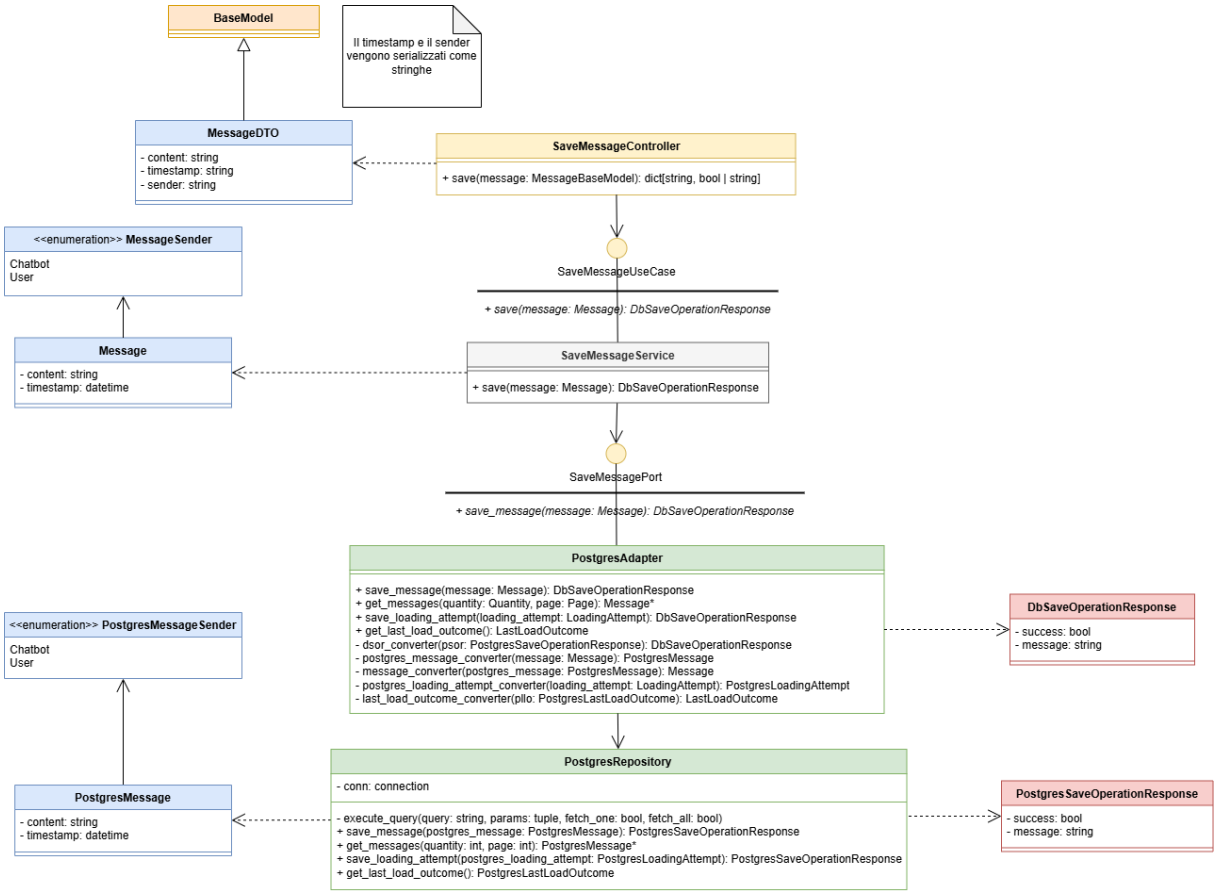

Figura 8: Architettura del salvataggio dei messaggi nello storico

L'UML in figura rappresenta l'architettura del salvataggio dei messaggi nello storico. Il *frontend* invoca il metodo `save_message` esposto dal backend tramite FastAPI, passando il messaggio in forma di MessageDTO. Il messaggio viene recuperato dal SaveMessageController, che converte il messaggio in un oggetto di Business, Message, e lo invia all'*interfaccia*$_G$ SaveMessageUseCase, implementata da SaveMessageService. SaveMessageService, quindi, chiama l'interfaccia SaveMessagePort, che è implementata da PostgresAdapter. PostgresAdapter si occupa di convertire il tipo di dato Message nel tipo PostgresMessage, e invoca il metodo `save_message` esposto dalla classe PostgresRepository. PostgresRepository gestisce il collegamento con il *database relazionale*$_G$ *Postgres*$_G$, esegue l'operazione di salvataggio del messaggio sul database, e restituisce un oggetto di tipo PostgresSaveOperationResponse che indica il successo o fallimento dell'operazione di salvataggio e la descrizione del successo o dell'eventuale errore. Successivamente, PostgresAdapter si occupa anche di convertire l'oggetto PostgresSaveOperationResponse in un oggetto DbSaveOperationResponse corrispondente e lo restituisce a SaveMessageService. Infine SaveMessageController riceve la DbSaveOperationResponse e la converte in un dizionario da restituire al frontend, che gestirà appositamente il successo o l'errore.

3.4.8 Architettura del recupero dei messaggi dallo storico

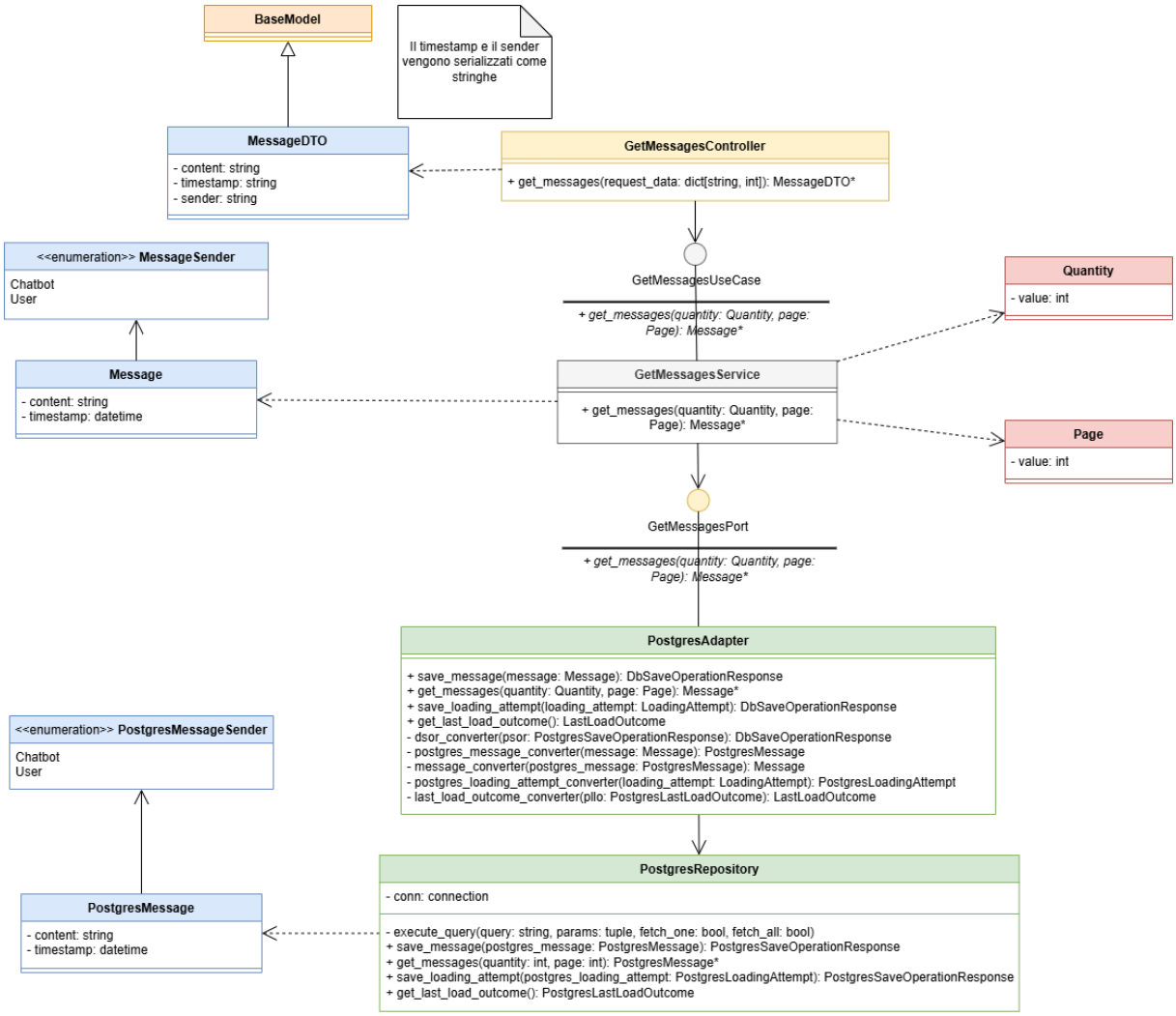

Figura 9: Architettura del recupero dei messaggi dallo storico

L'UML in figura rappresenta l'architettura del recupero dei messaggi nello storico. Il frontend invoca il metodo `get_messages` esposto dal backend tramite FastAPI, passando come parametro un dizionario che indica la quantità di messaggi da ottenere e il numero della pagina da cui ricavarli. Il metodo `get_messages` chiama GetMessagesController, che converte il dizionario in un oggetto Quantity e in un oggetto Page, e li passa all'interfaccia GetMessagesUseCase, implementata da GetMessagesService. GetMessagesService, quindi, chiama l'interfaccia GetMessagesPort, che è implementata da PostgresAdapter. PostgresAdapter converte gli oggetti Quantity e Page in interi da passare come parametri alla classe PostgresRepository, che gestisce il collegamento con il database relazionale Postgres. PostgresRepository esegue l'operazione di recupero della quantità di messaggi richiesti appartenenti alla pagina richiesta, e restituisce una lista di messaggi di tipo PostgresMessage. Allora PostgresAdapter si occupa di convertirla in una lista di oggetti Message, che viene restituita all'indietro a GetMessagesService. Infine GetMessagesController riceve una lista di Message e la usa per creare una lista di MessageDTO che restituisce al frontend.

3.4.9 Architettura della generazione di domande per proseguire la conversazione

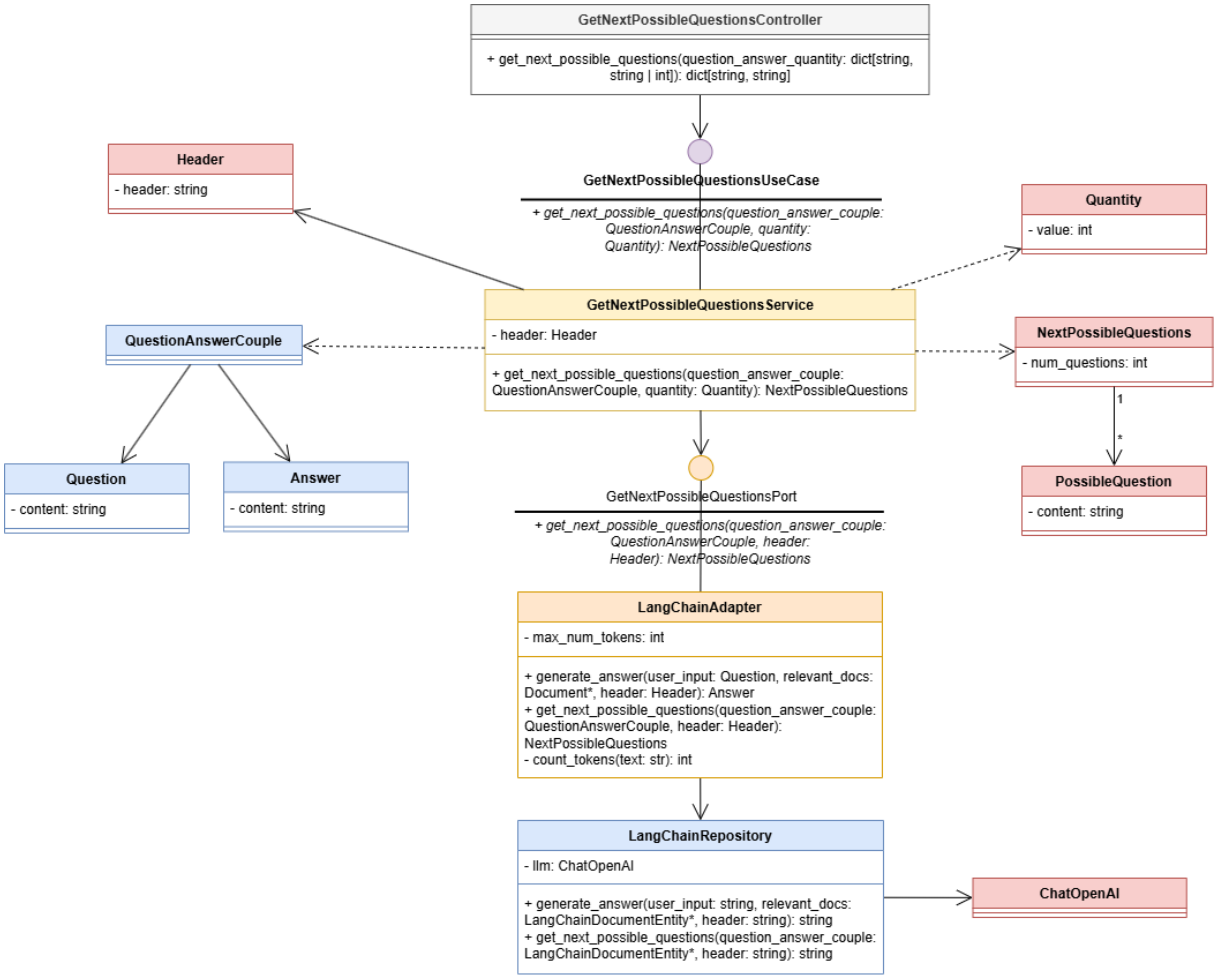

Figura 10: Architettura della generazione di domande per proseguire la conversazione

L'UML in figura rappresenta l'architettura della generazione di domande per proseguire la conversazione. Il frontend invoca il metodo `get_next_possible_questions` esposto dal backend tramite FastAPI, passando come parametro un dizionario che contiene l'ultima domanda dell'utente, l'ultima risposta del chatbot e la quantità di domande da ottenere. Il metodo `get_next_possible_questions` chiama GetNextPossibleQuestionsController, che converte il dizionario in un oggetto QuestionAnswerCouple contenente domanda (di tipo Question) e risposta (di tipo Answer) e in un oggetto Quantity contenente la quantità richiesta, e lo passa all'interfaccia GetNextPossibleQuestionsUseCase, implementata da GetNextPossibleQuestionsService. GetNextPossibleQuestionsService possiede un attributo di tipo Header contenente un'introduzione di contesto per l'LLM_G, templatizzata sulla base del numero di domande da richiedere, che a questo punto viene stabilito in base al parametro Quantity. L'attributo di tipo Header contiene anche importanti istruzioni di sintassi che l'LLM deve rispettare, cioè impone che le possibili domande generate siano l'unico contenuto del messaggio e siano separate da "__". La classe GetNextPossibleQuestionsService chiama dunque l'interfaccia GetNextPossibleQuestionsPort con parametri QuestionAnswerCouple ed Header. L'interfaccia è implementata da LangChainAdapter, che si occupa di adattare i parametri da QuestionAnswerCouple ed Header a, rispettivamente, una lista di oggetti LangChainDocumentEntity e una stringa, per poi inviarli a LangChainRepository, che si occupa di invocare l'LLM configurato per ottenere le possibili domande successive, in forma di un'unica stringa. Quest'ultima, in LangChainAdapter, viene suddivisa in 3 oggetti PossibleQuestion, sfruttando il fatto che l'header ha imposto all'LLM che le domande siano separate da "__", e, insieme ad un intero num_questions che indica il numero di domande generate, vengono poi riunite in un oggetto NextPossibleQuestions, per essere quindi restituite a GetNextPossibleQuestionsService. Infine, GetNextPossibleQuestionsController riceve l'oggetto NextPossibleQuestions, converte le domande contenute in un dizionario di stringhe, e lo restituisce al frontend.

3.5 Design pattern utilizzati

3.5.1 Facade

Il pattern *Facade*$_G$ è stato adottato per permettere al *cron*$_G$ di accedere ad unica classe LoadFilesService avente il compito di caricare tutti i tipi di file necessari per l'aggiornamento del database vettoriale. Questo pattern permette di nascondere la complessità del sistema, fornendo un'interfaccia semplificata per l'accesso alle funzionalità offerte dalla classe LoadFilesService, cioè caricamento da *GitHub*$_G$, *Jira*$_G$ e *Confluence*$_G$ verso *Chroma*$_G$ e *Postgres*$_G$. In questo modo, il cron non deve preoccuparsi di come i file vengono caricati, ma può semplicemente chiamare il metodo `load` di LoadFilesController che chiama il metodo `load` di LoadFilesService per ottenere il risultato desiderato.
E' stato inoltre adottato un piccolo Facade anche per l'architettura di generazione di una risposta, dove ChatService si occupa di nascondere la complessità della logica di business, fornendo un'interfaccia semplificata per l'accesso alle funzionalità offerte da SimilaritySearchService e GenerateAnswerService.
Grazie a questo pattern, è possibile mantenere il codice più pulito e manutenibile, in quanto la complessità è nascosta all'esterno e le interazioni con le classi interne sono semplificate.

3.5.2 Dependency Injection

Il pattern *Dependency Injection*$_G$ è stato adottato per permettere l'iniezione delle dipendenze all'interno delle classi del sistema. Questo pattern permette di separare la creazione degli oggetti dal loro utilizzo, semplificando la gestione delle dipendenze e rendendo il codice più manutenibile e testabile. In questo modo, le classi non devono preoccuparsi di come vengono creati gli oggetti di cui hanno bisogno, ma possono semplicemente riceverli come parametri di costruzione. Questo permette di sostituire facilmente le implementazioni delle dipendenze, senza dover modificare il codice di tutte le classi che le utilizzano. Inoltre, grazie all'utilizzo di un *framework*$_G$ di *Dependency Injection*, come *Angular*$_G$, è possibile automatizzare il processo di creazione e iniezione delle dipendenze, semplificando ulteriormente lo sviluppo del *frontend*$_G$. Il pattern è stato implementato anche in *Python*$_G$ nel *backend*$_G$, attraverso due funzioni, `dependency_injection_frontend` e `dependency_injection_cron`, che si occupano di iniettare le dipendenze rispettivamente nel file di endpoint *FastAPI*$_G$ a cui accede il frontend per contattare il backend e nello script che viene eseguito periodicamente dal cron per l'aggiornamento del *database vettoriale*$_G$.

3.5.3 MVVM

Il pattern *MVVM*$_G$ è stato adottato per la progettazione dell'architettura del *frontend*, in particolare per separare la logica di presentazione dalla logica di business, semplificando la gestione delle interazioni tra i componenti dell'applicazione. Nello specifico, il *Model*$_G$ rappresenta i dati e la logica di business dell'applicazione, il *View*$_G$ si occupa della presentazione dei dati all'utente, e il *ViewModel*$_G$ si occupa di gestire la logica di presentazione e di coordinare le interazioni tra il *Model* e la *View*. Questo permette di mantenere il codice più pulito e manutenibile, in quanto la logica di business è separata dalla logica di presentazione, e le interazioni tra i componenti sono semplificate. Inoltre, grazie all'utilizzo di un *framework* come *Angular*, è possibile automatizzare il processo di creazione e gestione dei componenti, semplificando ulteriormente lo sviluppo del *frontend*. Il pattern è stato implementato in *Angular* attraverso la creazione di componenti per la *View* e il *ViewModel*, e la creazione di servizi per la gestione della logica di business del *Model*.

4 Descrizione delle classi

4.1 Backend

4.1.1 Controller

4.1.1.1 ChatController

La classe ChatController si occupa di mettere in contatto il frontend con il backend nell'architettura di generazione di una risposta. Attraverso il metodo `get_answer(user_input: Request): dict[string, string]`, riceve il messaggio dell'utente di tipo Request, dal quale ricava un dizionario, che poi converte in un oggetto di Business, Question, e lo invia all'interfaccia ChatUseCase. Riceve poi da essa in output un oggetto Answer, lo converte in stringa, lo inserisce in un dizionario e infine lo restituisce al frontend.

4.1.1.2 LoadFilesController

La classe LoadFilesController si occupa di mettere in contatto il cron con il backend nell'architettura dell'aggiornamento automatico del database vettoriale. Attraverso il metodo `load()`, invia una richiesta all'interfaccia LoadFilesUseCase per il caricamento dei file da *GitHub*$_G$, *Jira*$_G$ e *Confluence*$_G$ verso *Chroma*$_G$, e il salvataggio dei rispettivi log in *Postgres*$_G$ e in un file di testo.

4.1.1.3 GetMessagesController

La classe GetMessagesController si occupa di mettere in contatto il frontend con il backend nell'architettura del recupero dei messaggi dallo storico. Attraverso il metodo `get_messages(request_data: dict[string, int]): MessageDTO*`, riceve un dizionario che indica la quantità di messaggi ed il numero di pagina da recuperare, lo converte in un oggetto Quantity, e lo invia all'interfaccia GetMessagesUseCase. Riceve poi da essa in output una lista di messaggi, la converte in una lista di MessageDTO e la restituisce al frontend.

4.1.1.4 SaveMessageController

La classe SaveMessageController si occupa di mettere in contatto il frontend con il backend nell'architettura del salvataggio dei messaggi nello storico. Attraverso il metodo `save(message: MessageDTO): dict[string, bool | string]`, riceve il messaggio dell'utente di tipo MessageDTO, lo converte in un oggetto di Business, Message, e lo invia all'interfaccia SaveMessageUseCase. Riceve poi da essa in output un DbSaveOperationResponse, lo converte in un dizionario e lo restituisce al frontend.

4.1.1.5 GetNextPossibleQuestionsController

La classe GetNextPossibleQuestionsController si occupa di mettere in contatto il frontend con il backend nell'architettura della generazione di domande per proseguire la conversazione. Attraverso il metodo `get_next_possible_questions(question_answer_quantity: dict[string, string | int]): dict[string, string]`, riceve un dizionario che contiene l'ultima domanda dell'utente, l'ultima risposta del chatbot e un intero che indica il numero di domande da generare, li converte in un oggetto QuestionAnswerCouple (contenente un oggetto Question e un oggetto Answer) e un oggetto Quantity, e li invia all'interfaccia GetNextPossibleQuestionsUseCase. Riceve poi da essa in output un oggetto NextPossibleQuestions, converte le domande contenute in un dizionario di stringhe, e lo restituisce al frontend.

4.1.1.6 GetLastLoadOutcomeController

La classe GetLastLoadOutcomeController si occupa di mettere in contatto il frontend con il backend nell'architettura dell'aggiornamento del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico. Attraverso il metodo `get_last_load_outcome(): LastLoadOutcomeDTO`, invia una richiesta all'interfaccia GetLastLoadOutcomeUseCase per ottenere l'esito dell'ultimo aggiornamento del database vettoriale, converte l'oggetto LastLoadOutcome ottenuto in un oggetto LastLoadOutcomeDTO e lo restituisce al frontend.

4.1.2 Use Case

4.1.2.1 ChatUseCase

ChatUseCase è un'interfaccia che definisce il caso d'uso della risposta ad una domanda dell'utente. Essa contiene un solo metodo, `get_answer(question: Question): Answer`, che prende in input una domanda, Question, e restituisce una risposta, Answer. Questa interfaccia fa da attributo a ChatController e viene implementata da ChatService.

4.1.2.2 LoadFilesUseCase

LoadFilesUseCase è un'interfaccia che definisce il caso d'uso dell'aggiornamento automatico del database vettoriale. Essa contiene un solo metodo, `load()`, la cui implementazione avvia il caricamento dei documenti nel *database vettoriale*$_G$. Questa interfaccia fa da attributo a LoadFilesController e viene implementata da LoadFilesService.

4.1.2.3 GetMessagesUseCase

GetMessagesUseCase è un'interfaccia che definisce il caso d'uso del recupero dei messaggi dallo storico. Essa contiene un solo metodo, `get_messages(quantity: Quantity, page: Page): Message*`, che prende in input la quantità dei messaggi da recuperare, di tipo Quantity, e il numero di pagina da recuperare, di tipo Page, e restituisce una lista di messaggi di tipo Message. Questa interfaccia fa da attributo a GetMessagesController e viene implementata da GetMessagesService.

4.1.2.4 SaveMessageUseCase

SaveMessageUseCase è un'interfaccia che definisce il caso d'uso del salvataggio dei messaggi nello storico. Essa contiene un solo metodo, `save(message: Message): DbSaveOperationResponse`, che prende in input un messaggio, Message, e restituisce una risposta dal database, DbSaveOperationResponse. Questa interfaccia fa da attributo a SaveMessageController e viene implementata da SaveMessageService.

4.1.2.5 GetNextPossibleQuestionsUseCase

GetNextPossibleQuestionsUseCase è un'interfaccia che definisce il caso d'uso della generazione di domande per proseguire la conversazione. Essa contiene un solo metodo, `get_next_possible_questions (question_answer_couple: QuestionAnswerCouple, quantity: Quantity): NextPossibleQuestions`, che prende in input una coppia domanda-risposta, QuestionAnswerCouple, e un oggetto di tipo Quantity contenente un intero che indica il numero di domande da generare, e restituisce le possibili domande successive, NextPossibleQuestions. Questa interfaccia fa da attributo a GetNextPossibleQuestionsController e viene implementata da GetNextPossibleQuestionsService.

4.1.2.6 GetLastLoadOutcomeUseCase

GetLastLoadOutcomeUseCase è un'interfaccia che definisce il caso d'uso dell'aggiornamento del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico. Essa contiene un solo metodo, `get_last_load_outcome(): LastLoadOutcome`, che restituisce l'esito dell'ultimo aggiornamento del database vettoriale, di tipo LastLoadOutcome. Questa interfaccia fa da attributo a GetLastLoadOutcomeController e viene implementata da GetLastLoadOutcomeService.

4.1.3 Service

4.1.3.1 ChatService

La classe ChatService implementa l'interfaccia ChatUseCase e si occupa di eseguire la logica di business per la generazione di una risposta all'interrogazione di un utente. Essa espone i seguenti tre metodi:

- `get_answer(question: Question): Answer`: Riceve in input un oggetto di Business, Question, e lo invia a SimilaritySearchService per la ricerca di similarità nel database vettoriale. Riceve in output i documenti rilevanti di tipo Document, li invia a GenerateAnswerService insieme all'input dell'utente, e infine restituisce la risposta di tipo Answer alla classe ChatController, di cui è attributo;
- `similarity_search(user_input: Question): Document*`: Riceve in input un oggetto di Business, Question, e lo invia a SimilaritySearchService per la ricerca di similarità nel database vettoriale. Restituisce i documenti rilevanti di tipo Document;
- `generate_answer(user_input: Question, relevant_docs: Document*): Answer`: Riceve in input un oggetto di Business, Question, e i documenti rilevanti di tipo Document, e li invia a GenerateAnswerService per la generazione di una risposta. Restituisce la risposta di tipo Answer.

4.1.3.2 SimilaritySearchService

La classe SimilaritySearchService si occupa di eseguire la ricerca di *similarità*$_G$ nel database vettoriale. Essa espone un solo metodo, `similarity_search(user_input: Question): Document*`, che prende in input un oggetto di Business, Question, e chiama l'interfaccia SimilaritySearchPort per svolgere la ricerca di similarità in un database vettoriale. La porta restituisce una grande quantità di documenti rilevanti di tipo Document, i quali vengono sottoposti ad un filtraggio utilizzando un attributo di tipo DocumentConstraints (che possiede un attributo threshold e un attributo max_gap), che impone che la distanza di similarità dei documenti dalla domanda rispetti un threshold, e impone che, se un documento è distante dal successivo elemento della lista di più del max_gap, tutti i documenti successivi non passino il filtro. Infine, i documenti filtrati vengono restituiti a ChatService.

4.1.3.3 GenerateAnswerService

La classe GenerateAnswerService si occupa di generare una risposta all'interrogazione di un utente. Essa espone un solo metodo, `generate_answer(user_input: Question, relevant_docs: Document*): Answer`, che prende in input un oggetto di Business, Question, e i documenti rilevanti di tipo Document, e ci associa l'oggetto di tipo Header che possiede come attributo, per poter così chiamare l'interfaccia GenerateAnswerPort allo scopo di generare una risposta. La risposta, di tipo Answer, viene restituita a ChatService.

4.1.3.4 LoadFilesService

La classe LoadFilesService implementa l'interfaccia LoadFilesUseCase e si occupa di eseguire la logica di business per l'aggiornamento automatico del database vettoriale. Essa implementa il *design pattern*$_G$ *Facade*$_G$, ed espone i seguenti metodi:

- `load()`: Si occupa di chiamare i metodi di estrazione dei file da GitHub, Jira e Confluence, di pulizia delle pagine Confluence, di caricamento in Chroma e di salvataggio dei log in Postgres e in un file di testo;
- `load_github_commits(): (PlatformLog, Document*)`: Si occupa di recuperare i Commits da GitHub e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti ai commits;
- `load_github_files(): (PlatformLog, Document*)`: Si occupa di recuperare i Files da GitHub e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti ai files;
- `load_jira_issues(): (PlatformLog, Document*)`: Si occupa di recuperare le Issues da Jira e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti alle issues;

- `load_confluence_pages(): (PlatformLog, Document*)`: Si occupa di recuperare le Pagine da Confluence e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti alle pagine;
- `get_github_files_new_metadata(github_file: Document*, github_commit: Document*): Document*`: Si occupa di riempire i campi 'last_update' e 'creation_date' dei metadati dei file di GitHub, basandosi sul confronto con i GitHub commit in cui il singolo file è stato coinvolto e lo status con cui è stato coinvolto ('added', 'modified' o 'renamed'). Restituisce la lista di GitHub file con i metadati aggiornati, sempre di tipo Document;
- `clean_confluence_pages(pages: Document*): Document*`: Si occupa di pulire le pagine di Confluence, rimuovendo i tag HTML e i caratteri speciali. Restituisce la lista di pagine pulite, sempre di tipo Document;
- `load_in_vector_store(documents: Document*): VectorStoreLog`: Si occupa di caricare i documenti nel database vettoriale Chroma e di salvare i rispettivi log. Restituisce un oggetto di tipo VectorStoreLog, corrispondente al log del caricamento;
- `save_loading_attempt_in_db(loading_attempt: LoadingAttempt): DbSaveOperationResponse`: Si occupa di salvare l'esito del caricamento dei documenti nel database relazionale Postgres. Restituisce un oggetto di tipo DbSaveOperationResponse, corrispondente alla risposta del database;
- `save_loading_attempt_in_txt(loading_attempt: LoadingAttempt)`: Si occupa di salvare l'esito del caricamento dei documenti in un file di testo, sfruttando la libreria `logging` di *Python*$_G$.

4.1.3.5 ConfluenceCleanerService

La classe ConfluenceCleanerService si occupa di pulire le pagine di Confluence, rimuovendo i tag HTML e i caratteri speciali. Essa espone il seguente metodo pubblico:

- `clean_confluence_pages(pages: Document*): Document*`: Si occupa di pulire le pagine di Confluence, rimuovendo i tag HTML e i caratteri speciali. Restituisce la lista di pagine pulite, sempre di tipo Document.

Inoltre, contiene i seguenti metodi privati:

- `remove_html_tags(document: Document): Document`: Si occupa di rimuovere i tag HTML da una pagina di Confluence. Restituisce la pagina pulita, di tipo Document;
- `replace_html_entities(document: Document): Document`: Si occupa di sostituire le entità HTML con i corrispondenti caratteri Unicode da una pagina di Confluence. Restituisce la pagina pulita, di tipo Document.

4.1.3.6 GetMessagesService

La classe GetMessagesService implementa l'interfaccia GetMessagesUseCase e si occupa di eseguire la logica di business per il recupero dei messaggi dallo storico. Essa espone un solo metodo, `get_messages (quantity: Quantity, page: Page): Message*`, che prende in input la quantità dei messaggi da recuperare, di tipo Quantity, e il numero di pagina da recuperare, di tipo Page, e chiama l'interfaccia GetMessagesPort per recuperare la quantità di messaggi richiesta nella pagina richiesta. La porta restituisce una lista di messaggi, che vengono a loro volta restituiti a GetMessagesController.

4.1.3.7 SaveMessageService

La classe SaveMessageService implementa l'interfaccia SaveMessageUseCase e si occupa di eseguire la logica di business per il salvataggio dei messaggi nello storico. Essa espone un solo metodo, `save(message: Message): DbSaveOperationResponse`, che prende in input un oggetto di Business, Message, e chiama l'interfaccia SaveMessagePort per salvare il messaggio sul database. La porta restituisce un oggetto di Business, DbSaveOperationResponse, che viene a sua volta restituito a SaveMessageController.

4.1.3.8 GetNextPossibleQuestionsService

La classe GetNextPossibleQuestionsService implementa l'interfaccia GetNextPossibleQuestionsUseCase e si occupa di eseguire la logica di business per la generazione di domande per proseguire la conversazione. Essa espone un solo metodo, `get_next_possible_questions(question_answer_couple: QuestionAnswerCouple, quantity: Quantity): NextPossibleQuestions`, che prende in input una coppia domanda-risposta, QuestionAnswerCouple, un oggetto di tipo Quantity contenente un intero che indica il numero di domande da generare, e ci associa l'oggetto di tipo Header che possiede come attributo, per poter così chiamare l'interfaccia GetNextPossibleQuestionsPort per generare le possibili domande successive. La porta restituisce le possibili domande racchiuse in un oggetto di Business, NextPossibleQuestions, che viene a sua volta restituito a GetNextPossibleQuestionsController.

4.1.3.9 GetLastLoadOutcomeService

La classe GetLastLoadOutcomeService implementa l'interfaccia GetLastLoadOutcomeUseCase e si occupa di eseguire la logica di business per l'aggiornamento del badge di segnalazione dell'esito dell'ultimo aggiornamento automatico. Essa espone un solo metodo, `get_last_load_outcome(): LastLoadOutcome`, che chiama l'interfaccia GetLastLoadOutcomePort per ottenere l'esito dell'ultimo aggiornamento del database vettoriale. La porta restituisce un oggetto di Business, LastLoadOutcome, che viene a sua volta restituito a GetLastLoadOutcomeController.

4.1.4 Port

4.1.4.1 SimilaritySearchPort

SimilaritySearchPort definisce l'interfaccia attraverso la quale SimilaritySearchService interagisce con il mondo esterno, permettendo il recupero di dati persistenti senza modificare la logica di business. Essa espone un solo metodo,
`similarity_search(user_input: Question): Document*`, che prende in input la domanda dell'utente di tipo Question, e si occupa di svolgere la ricerca di similarità in un *database vettoriale*$_G$ per ottenere una lista di documenti rilevanti di tipo Document. La porta viene implementata da ChromaVectorStoreAdapter.

4.1.4.2 GenerateAnswerPort

GenerateAnswerPort definisce l'interfaccia attraverso la quale GenerateAnswerService interagisce con il mondo esterno, permettendo la generazione di dati da parte di un *LLM*$_G$ senza modificare la logica di business. Essa espone un solo metodo,
`generate_answer(user_input: Question, relevant_docs: Document*, header: Header): Answer`, che prende in input la domanda dell'utente di tipo Question, i documenti rilevanti di tipo Document, e un oggetto Header che fornisce il contesto introduttivo al chatbot, e si occupa di generare una risposta di tipo Answer. La porta viene implementata da LangChainAdapter.

4.1.4.3 GitHubPort

GitHubPort definisce una delle interfacce attraverso le quali LoadFilesService interagisce con il mondo esterno, permettendo il recupero di dati persistenti senza modificare la logica di business. Essa espone due metodi:

- `load_github_commits(): (PlatformLog, Document*)`: Si occupa di recuperare i Commits da GitHub e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti ai commits;
- `load_github_files(): (PlatformLog, Document*)`: Si occupa di recuperare i Files da GitHub e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti ai files.

La porta viene implementata da GitHubAdapter.

4.1.4.4 JiraPort

JiraPort definisce una delle interfacce attraverso le quali LoadFilesService interagisce con il mondo esterno, permettendo il recupero di dati persistenti senza modificare la logica di business. Essa espone un solo metodo, `load_jira_issues(): (PlatformLog, Document*)`, che si occupa di recuperare le Issues da Jira e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti alle issues. La porta viene implementata da JiraAdapter.

4.1.4.5 ConfluencePort

ConfluencePort definisce una delle interfacce attraverso le quali LoadFilesService interagisce con il mondo esterno, permettendo il recupero di dati persistenti senza modificare la logica di business. Essa espone un solo metodo, `load_confluence_pages(): (PlatformLog, Document*)`, che si occupa di recuperare le Pagine da Confluence e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti alle pagine. La porta viene implementata da ConfluenceAdapter.

4.1.4.6 LoadFilesInVectorStorePort

LoadFilesInVectorStorePort definisce una
delle interfacce attraverso le quali LoadFilesService interagisce con il mondo esterno, permettendo il caricamento dei documenti nel database vettoriale senza modificare la logica di business. Essa espone un solo metodo, `load(documents: Document*): VectorStoreLog`, che prende in input una lista di documenti

di tipo Document, e si occupa di caricare i documenti nel database vettoriale configurato e di restituito il corrispondente log, di tipo VectorStoreLog. La porta viene implementata da ChromaVectorStoreAdapter.

4.1.4.7 SaveLoadingAttemptInDbPort

SaveLoadingAttemptInDbPort definisce una delle interfacce attraverso le quali LoadFilesService interagisce con il mondo esterno, permettendo il salvataggio di dati persistenti senza modificare la logica di business. Essa espone un solo metodo, `save_loading_attempt(loading_attempt: LoadingAttempt): DbSaveOperationResponse`, che prende in input l'esito del caricamento, LoadingAttempt, e si occupa di salvarlo nel database relazionale configurato e di restituire la corrispondente risposta, di tipo DbSaveOperationResponse. La porta viene implementata da PostgresAdapter.

4.1.4.8 GetMessagesPort

GetMessagesPort definisce l'interfaccia attraverso la quale GetMessagesService interagisce con il mondo esterno, permettendo il recupero di dati persistenti senza modificare la logica di business. Essa espone un solo metodo, `get_messages(quantity: Quantity, page: Page): Message*`, che prende in input la quantità dei messaggi da recuperare, di tipo Quantity, e il numero di pagina da recuperare, di tipo Page, e si occupa di recuperare una lista di messaggi, di tipo Message, da un database. La porta viene implementata da PostgresAdapter.

4.1.4.9 SaveMessagePort

SaveMessagePort definisce l'interfaccia attraverso la quale SaveMessageService interagisce con il mondo esterno, permettendo il salvataggio di dati persistenti senza modificare la logica di business. Essa espone un solo metodo, `save(message: Message): DbSaveOperationResponse`, che prende in input il messaggio da salvare di tipo Message, e si occupa di salvare il messaggio in un database e riportare la sua risposta, di tipo DbSaveOperationResponse. La porta viene implementata da PostgresAdapter.

4.1.4.10 GetNextPossibleQuestionsPort

GetNextPossibleQuestionsPort definisce l'interfaccia attraverso la quale GetNextPossibleQuestionsService interagisce con il mondo esterno, permettendo la generazione di possibili domande successive senza modificare la logica di business. Essa espone un solo metodo, `get_next_possible_questions (question_answer_couple: QuestionAnswerCouple, header: Header): NextPossibleQuestions`, che prende in input una coppia domanda-risposta, QuestionAnswerCouple, e un'introduzione per dare contesto al chatbot, Header, e si occupa di generare le possibili domande successive, di tipo NextPossibleQuestions. La porta viene implementata da LangChainAdapter.

4.1.4.11 GetLastLoadOutcomePort

GetLastLoadOutcomePort definisce l'interfaccia attraverso la quale GetLastLoadOutcomeService interagisce con il mondo esterno, permettendo il recupero di dati persistenti senza modificare la logica di business. Essa espone un solo metodo, `get_last_load_outcome(): LastLoadOutcome`, che si occupa di ottenere l'esito dell'ultimo aggiornamento del database vettoriale, di tipo LastLoadOutcome. La porta viene implementata da PostgresAdapter.

4.1.5 Adapter

4.1.5.1 ChromaVectorStoreAdapter

La classe ChromaVectorStoreAdapter implementa le interfacce LoadFilesInVectorStorePort e SimilaritySearchPort, e si occupa di adattare i dati ricevuti dalla logica di business in un tipo Entity adatto per la comunicazione con il mondo esterno, e viceversa. Essa possiede, oltre a chroma_vector_store_repository, un altro attributo privato, ricevuto tramite il costruttore, che viene chiamato da una funzione di *Dependency Injection*$_G$:

- `max_chunk_size: int`: Indica la dimensione massima di un *chunk*$_G$ di documenti che è possibile caricare nel database vettoriale. Per Chroma, questo valore equivale a 41.666 caratteri.

Essa espone i seguenti due metodi pubblici:

- `load(documents: Document*): VectorStoreLog`: Riceve in input una lista di documenti di tipo Document, chiama il proprio metodo split per dividerli in chunk e convertirli in oggetti ChromaDocumentEntity, e si occupa di chiamare il metodo di ChromaVectorStoreRepository per caricare i documenti nel database vettoriale. Restituisce un oggetto di tipo VectorStoreLog, corrispondente al log del caricamento;
- `similarity_search(user_input: Question): Document*`: Riceve in input la domanda dell'utente di tipo Question, la converte in una stringa, e si occupa di chiamare il metodo di ChromaVectorStoreRepository per cercare i documenti similari alla domanda nel database vettoriale. Da tal metodo riceve un oggetto QueryResultEntity, che converte in una lista di documenti di tipo Document, corrispondenti ai documenti rilevanti. Questi ultimi vengono restituiti a SimilaritySearchService.

Essa contiene inoltre il seguente metodo privato:

- `split(documents: Document*): ChromaDocumentEntity*`: Riceve in input una lista di documenti di tipo Document, li converte in oggetti DocumentEntity, e si occupa di dividere i documenti in chunk di dimensione massima max_chunk_size. Restituisce una lista di oggetti di tipo ChromaDocumentEntity, corrispondenti ai chunk di documenti.

4.1.5.2 LangChainAdapter

La classe LangChainAdapter implementa le interfacce GenerateAnswerPort e GetNextPossibleQuestionsPort, e si occupa di adattare i dati ricevuti dalla logica di business in un tipo Entity adatto per la comunicazione con il mondo esterno, e viceversa. Essa possiede, oltre a langchain_repository, un altro attributo privato, ricevuto tramite il costruttore, che viene chiamato da una funzione di Dependency Injection:

- `max_num_tokens: int`: Indica il numero massimo di *token*$_G$ che è possibile utilizzare per generare una risposta. Per l'LLM *OpenAI*$_G$ GPT-4o, questo valore equivale a 128.000 token.

Essa espone i seguenti due metodi pubblici:

- `generate_answer(user_input: Question, relevant_docs: Document*, header: Header): Answer`: Riceve in input la domanda dell'utente di tipo Question, che converte in una stringa, una lista di documenti rilevanti di tipo Document, che converte in una lista di LangChainDocumentEntity (occupandosi anche di calcolare il numero di token del contenuto dei documenti tramite il metodo `count_tokens` e di escludere alcuni documenti per non oltrepassare il `max_num_tokens`), e un oggetto Header che fornisce il contesto introduttivo al chatbot, che converte in una stringa, e si occupa di chiamare il metodo di LangChainRepository per generare una risposta. Riceve da quest'ultimo una risposta di tipo stringa, la converte in un tipo Answer e la restituisce a GenerateAnswerService;
- `get_next_possible_questions(question_answer_couple: QuestionAnswerCouple, header: Header): NextPossibleQuestions`: Riceve in input una coppia domanda-risposta di tipo QuestionAnswerCouple (contenente una Question e una Answer), che converte in una lista di documenti LangChainDocumentEntity, e un'introduzione per dare contesto al chatbot di tipo Header, che converte in una stringa, e si occupa di chiamare il metodo di LangChainRepository per generare le possibili domande successive. Riceve da quest'ultimo una stringa, che, grazie alle istruzioni fornite dall'header, possiederà unicamente le domande e le possiederà suddivise da "__".

Quindi, effettua il parsing della stringa ed estrapola le domande per crearci una lista di oggetti PossibleQuestion, che poi, assieme all'intero num_questions, il cui valore coincide con la lunghezza della lista, vengono incapsulate in un oggetto NextPossibleQuestions, che viene restituito a GetNextPossibleQuestionsService.

Inoltre, contiene il seguente metodo privato:

- `count_tokens(text: str): int`: Riceve in input un testo di tipo stringa, e si occupa di contare il numero di token presenti in esso. Si basa sull'assunzione che 1 token equivalga a 2 caratteri. Restituisce un intero, corrispondente al numero di token calcolato.

4.1.5.3 GitHubAdapter

La classe GitHubAdapter implementa l'interfaccia GitHubPort e si occupa di adattare i dati ricevuti dalla logica di business in un tipo Entity adatto per la comunicazione con il mondo esterno, e viceversa. Essa espone i seguenti due metodi pubblici:

- `load_github_commits(): (PlatformLog, Document*)`: Si occupa di chiamare il metodo di GitHubRepository per recuperare i Commits da GitHub e restituire il rispettivo log. Dal repository riceve un PlatformLog e una lista di CommitEntity, e converte quest'ultima in una lista di Document. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti ai Commits;
- `load_github_files(): (PlatformLog, Document*)`: Si occupa di chiamare il metodo di GitHubRepository per recuperare i Files da GitHub e restituire il rispettivo log. Dal repository riceve un PlatformLog e una lista di FileEntity, e converte quest'ultima in una lista di Document. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti ai Files.

Inoltre, contiene il seguente metodo privato:

- `UTC_to_CET(timestamp: datetime): string`: Riceve in input un timestamp di tipo datetime contenente una data e ora secondo il fuso orario *UTC*, e si occupa di convertirlo in una stringa che rappresenta il timestamp convertito nel fuso orario *CET*. Restituisce una stringa contenente il timestamp convertito.

4.1.5.4 JiraAdapter

La classe JiraAdapter implementa l'interfaccia JiraPort e si occupa di adattare i dati ricevuti dalla logica di business in un tipo Entity adatto per la comunicazione con il mondo esterno, e viceversa. Essa espone il seguente metodo:

- `load_jira_issues(): (PlatformLog, Document*)`: Si occupa di chiamare il metodo di JiraRepository per recuperare le Issues da Jira e restituire il rispettivo log. Dal repository riceve un PlatformLog e una lista di IssueEntity, e converte quest'ultima in una lista di Document. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti alle issues.

4.1.5.5 ConfluenceAdapter

La classe ConfluenceAdapter implementa l'interfaccia ConfluencePort e si occupa di adattare i dati ricevuti dalla logica di business in un tipo Entity adatto per la comunicazione con il mondo esterno, e viceversa. Essa espone il seguente metodo pubblico:

- `load_confluence_pages(): (PlatformLog, Document*)`: Si occupa di chiamare il metodo di ConfluenceRepository per recuperare le Pagine da Confluence e restituire il rispettivo log. Dal repository riceve un PlatformLog e una lista di PageEntity, e converte quest'ultima in una lista di Document. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo Document, corrispondenti alle pagine.

Inoltre, contiene il seguente metodo privato:

- `UTC_to_CET(timestamp: string): string`: Riceve in input un timestamp di tipo stringa contenente una data e ora secondo il fuso orario *UTC*, e si occupa di convertirlo in una stringa che rappresenta il timestamp convertito nel fuso orario *CET*. Restituisce una stringa contenente il timestamp convertito.

4.1.5.6 PostgresAdapter

La classe PostgresAdapter implementa le interfacce GetMessagesPort, SaveMessagePort, SaveLoadingAttemptInDbPort e GetLastLoadOutcomePort, e si occupa di adattare i dati ricevuti dalla logica di business in un tipo Entity adatto per la persistenza, e viceversa. Essa espone i seguenti metodi pubblici:

- `get_messages(quantity: Quantity, page: Page): Message*`: Riceve in input la quantità dei messaggi da recuperare, di tipo Quantity, e il numero di pagina da recuperare, di tipo Page, e si occupa di convertirli in interi e di chiamare il metodo di recupero dei messaggi dal database di PostgresRepository. Riceve una lista di messaggi di tipo PostgresMessage, e restituisce, una volta opportunamente convertita, una lista di messaggi di tipo Message;
- `save_message(message: Message): DbSaveOperationResponse`: Riceve in input il messaggio da salvare di tipo Message, e si occupa di chiamare il metodo di conversione del messaggio in PostgresMessage e di chiamare il metodo di salvataggio di un messaggio nel database di PostgresRepository. Riceve un oggetto di tipo PostgresSaveOperationResponse e restituisce, una volta opportunamente convertito, un oggetto di tipo DbSaveOperationResponse;
- `save_loading_attempt(loading_attempt: LoadingAttempt): DbSaveOperationResponse`: Riceve in input il tentativo di caricamento da salvare di tipo LoadingAttempt, e si occupa di chiamare il metodo di conversione del tentativo in PostgresLoadingAttempt e di chiamare il metodo di salvataggio del tentativo di caricamento nel database di PostgresRepository. Riceve un oggetto di tipo PostgresSaveOperationResponse e restituisce, una volta opportunamente convertito, un oggetto di tipo DbSaveOperationResponse;
- `get_last_load_outcome(): LastLoadOutcome`: Si occupa di chiamare il metodo di recupero dell'esito dell'ultimo tentativo di caricamento di documenti nel database vettoriale da PostgresRepository. Riceve un oggetto di tipo PostgresLastLoadOutcome e restituisce, una volta opportunamente convertito, un oggetto di tipo LastLoadOutcome.

Inoltre, contiene i seguenti metodi privati:

- `postgres_message_converter(message: Message): PostgresMessage`: Riceve in input un messaggio di tipo Message, e si occupa di convertirlo in un messaggio di tipo PostgresMessage;
- `message_converter(postgres_message: PostgresMessage): Message`: Riceve in input un messaggio di tipo PostgresMessage, e si occupa di convertirlo in un messaggio di tipo Message;
- `dsor_converter(psor: PostgresSaveOperationResponse): DbSaveOperationResponse`: Riceve in input un oggetto di tipo PostgresSaveOperationResponse, e si occupa di convertirlo in un oggetto di tipo DbSaveOperationResponse;
- `postgres_loading_attempt_converter(loading_attempt: LoadingAttempt): PostgresLoadingAttempt`: Riceve in input un tentativo di caricamento di tipo LoadingAttempt, e si occupa di convertirlo in un tentativo di caricamento di tipo PostgresLoadingAttempt;
- `last_load_outcome_converter(pllo: PostgresLastLoadOutcome): LastLoadOutcome`: Riceve in input un oggetto di tipo PostgresLastLoadOutcome, e si occupa di convertirlo in un oggetto di tipo LastLoadOutcome.

4.1.6 Repository

4.1.6.1 ChromaVectorStoreRepository

La classe ChromaVectorStoreRepository si occupa di gestire parte della *persistent logic*$_G$ del sistema, ovvero di caricare i documenti nel database vettoriale *Chroma*$_G$ e di recuperarli compiendo una ricerca di *similarità*$_G$. Essa contiene un singolo attributo privato, ricevuto tramite il costruttore, che viene chiamato da una funzione di *Dependency Injection*:

- `collection: Collection`: Collezione di documenti del database vettoriale Chroma, chiamata "buddybot-vector-store".

Essa espone i seguenti metodi pubblici:

- `load(documents: ChromaDocumentEntity*): VectorStoreLog`: Si occupa di caricare i documenti nel database vettoriale Chroma. Riceve in input una lista di documenti di tipo ChromaDocumentEntity e li carica nel database vettoriale. Si occupa anche di distinguere chiaramente quali documenti sono stati aggiunti, quali sono stati modificati (cioè eliminati e riaggiunti aggiornati) e quali sono stati eliminati: viene considerata la data di ultimo aggiornamento per le Jira Issues, le Confluence Pages e i GitHub Commits, mentre avviene il confronto tra la data di creazione e la data di ultimo inserimento in Chroma per i GitHub Files, che hanno bisogno di questo passaggio dedicato poichè la piattaforma GitHub non fornisce un id univoco per distinguere un file da un altro senza dipendere dal contenuto. Restituisce un oggetto di tipo VectorStoreLog, corrispondente al log del caricamento riportante le suddette statistiche sui documenti caricati;

- `similarity_search(user_input: string): QueryResultEntity`: Si occupa di cercare i documenti similari alla domanda dell'utente nel database vettoriale Chroma. Riceve in input la domanda dell'utente di tipo string e la utilizza per cercare i documenti similari nel database vettoriale. Restituisce un oggetto di tipo QueryResultEntity, contenente i documenti rilevanti.

4.1.6.2 LangChainRepository

La classe LangChainRepository si occupa di gestire parte della *persistent logic*$_G$ del sistema, ovvero di generare risposte a partire da domande e documenti rilevanti, e di generare possibili domande successive a partire da una domanda e una risposta. Essa contiene un singolo attributo privato, ricevuto tramite il costruttore, che viene chiamato da una funzione di *Dependency Injection*:

- `llm: ChatOpenAI`: Modello di *LLM*$_G$ di *OpenAI*$_G$, più precisamente GPT-4o.

Essa espone i seguenti metodi pubblici:

- `generate_answer(user_input: string, relevant_docs: LangChainDocumentEntity*, header: string): string`: Si occupa di generare una risposta a partire dalla domanda dell'utente e dai documenti rilevanti. Riceve in input la domanda dell'utente di tipo string, i documenti rilevanti di tipo LangChainDocumentEntity e un'introduzione per dare contesto al chatbot di tipo string. Sfrutta le funzioni della libreria *LangChain*$_G$ per creare il prompt e contattare l'LLM. Restituisce una stringa, corrispondente alla risposta generata;

- `get_next_possible_questions(question_answer_couple: LangChainDocumentEntity*, header: string): string`: Si occupa di generare possibili domande successive a partire da una domanda e una risposta. Riceve in input una lista di documenti LangChainDocumentEntity contenenti la domanda dell'utente e la risposta ricevuta, e un'introduzione per dare contesto al chatbot di tipo string. Sfrutta le funzioni della libreria *LangChain* per creare il prompt e contattare l'LLM. Restituisce una stringa, contenente unicamente le possibili domande successive, separate da "__", come da istruzioni dell'header.

4.1.6.3 GitHubRepository

La classe GitHubRepository si occupa di gestire parte della *persistent logic*$_G$ del sistema, ovvero di recuperare i dati da *GitHub*$_G$. Essa contiene un singolo attributo privato:

- `github_repo: Repository`: Repository GitHub da cui recuperare i dati, ricevuto tramite il costruttore, che viene chiamato da una funzione di *Dependency Injection*.

Essa espone i seguenti metodi pubblici:

- `load_github_commits(): (PlatformLog, CommitEntity*)`: Si occupa di recuperare i Commits da GitHub e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo CommitEntity, corrispondenti ai commits;
- `load_github_files(): (PlatformLog, FileEntity*)`: Si occupa di recuperare i Files da GitHub e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo FileEntity, corrispondenti ai files.

4.1.6.4 JiraRepository

La classe JiraRepository si occupa di gestire parte della *persistent logic*$_G$ del sistema, ovvero di recuperare i dati da *Jira*$_G$. Essa contiene quattro attributi privati, ricevuti tramite il costruttore, che viene chiamato da una funzione di *Dependency Injection*:

- `base_url: string`: URL di base del profilo Jira del progetto;
- `project_key: string`: Chiave del progetto Jira;
- `timeout: int`: Timeout delle richieste;
- `headers: dict[string, string]`: Headers delle richieste, per gestire l'autenticazione.

Essa espone il seguente metodo:

- `load_jira_issues(): (PlatformLog, IssueEntity*)`: Si occupa di recuperare le Issues da Jira e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo IssueEntity, corrispondenti alle issues.

4.1.6.5 ConfluenceRepository

La classe ConfluenceRepository si occupa di gestire parte della *persistent logic*$_G$ del sistema, ovvero di recuperare i dati da *Confluence*$_G$. Essa contiene quattro attributi privati, ricevuti tramite il costruttore, che viene chiamato da una funzione di *Dependency Injection*:

- `base_url: string`: URL di base del profilo Confluence del progetto;
- `space_key: string`: Chiave dello spazio Confluence;
- `timeout: int`: Timeout delle richieste;
- `headers: dict[string, string]`: Headers delle richieste, per gestire l'autenticazione.

Essa espone il seguente metodo:

- `load_confluence_pages(): (PlatformLog, PageEntity*)`: Si occupa di recuperare le Pagine da Confluence e di salvare i rispettivi log. Restituisce una tupla contenente un oggetto di tipo PlatformLog, corrispondente al log, e una lista di oggetti di tipo PageEntity, corrispondenti alle pagine.

4.1.6.6 PostgresRepository

La classe PostgresRepository si occupa di gestire parte della *persistent logic*$_G$ del sistema, ovvero di gestire la persistenza dei dati, interagendo con il *database relazionale*$_G$ *Postgres*$_G$ per salvare e/o recuperare le informazioni necessarie. Essa contiene un singolo attributo privato, ricevuto tramite il costruttore, che viene chiamato da una funzione di *Dependency Injection*:

- `conn: Connection`: Connessione al database Postgres.

Essa espone i seguenti metodi pubblici:

- `get_messages(quantity: int, page: int): PostgresMessage*`: Riceve in input la quantità di messaggi e il numero di pagina, di tipo intero, e si occupa di recuperare tale quantità di messaggi di tale pagina dal database. Restituisce una lista di messaggi di tipo PostgresMessage;

- `save_message(message: PostgresMessage): PostgresSaveOperationResponse`: Riceve in input un messaggio di tipo PostgresMessage, e si occupa di salvarlo nel database. Restituisce un oggetto di tipo PostgresSaveOperationResponse, contenente l'esito dell'operazione e un relativo messaggio;
- `save_loading_attempt(loading_attempt: PostgresLoadingAttempt): PostgresSaveOperationResponse`: Riceve in input un tentativo di caricamento di tipo PostgresLoadingAttempt, e si occupa di salvarlo nel database. Restituisce un oggetto di tipo PostgresSaveOperationResponse, contenente l'esito dell'operazione e un relativo messaggio;
- `get_last_load_outcome(): PostgresLoadingAttempt`: Si occupa di recuperare l'esito dell'ultimo tentativo di caricamento di documenti nel database vettoriale. Più precisamente, viene recuperato l'attributo `outcome` della tupla con timestamp di fine aggiornamento più recente dalla tabella `loading_attempts`. Restituisce dunque un oggetto di tipo PostgresLastLoadOutcome, che può assumere valore True, se l'ultimo tentativo di aggiornamento ha avuto esito positivo, False, se l'ultimo tentativo di aggiornamento ha avuto esito negativo, o Error, se c'è stato un errore nell'interazione con Postgres per recuperare l'esito dell'ultimo aggiornamento.

Inoltre, contiene il seguente metodo privato:

- `execute_query(query: string, params: tuple, fetch_one: bool, fetch_all: bool): tuple | list | None`: Riceve in input una stringa contenente una query SQL_G, i parametri della query, e due booleani opzionali che indicano se la funzione deve restituire un solo risultato, tutti i risultati, oppure nessuno. Si occupa di eseguire la query sul database configurato e, eventualmente, di restituirne i risultati.

4.2 Frontend

4.2.1 Component

4.2.1.1 AppComponent

AppComponent rappresenta il componente contenitore di tutta l'applicazione Angular. Tuttavia, nel nostro caso, il suo ruolo si limita a quello di contenitore per il componente ChatContainerComponent, che poi sarà l'effettivo contenitore e "manager" dei vari componenti dell'applicazione.

4.2.1.2 ChatContainerComponent

ChatContainerComponent rappresenta il componente contenitore di tutti i componenti che compongono l'applicazione. Esso contiene i seguenti componenti:

- ChatHeaderComponent;
- ChatMessagesComponent;
- ChatSuggestionsComponent;
- ChatInputComponent.

ChatContainerComponent si occupa di gestire la logica di business della chat, ovvero di recuperare i messaggi, di inviarli, di visualizzarli e di gestire le possibili domande successive. Il componente espone i seguenti metodi:

- `ngOnInit()`: Si occupa di inizializzare il componente, recuperando i messaggi e l'esito dell'ultimo caricamento nel database vettoriale dal database;
- `loadOldMessages(quantity: number, page: number = 1)`: Si occupa di recuperare i messaggi dello storico della chat, di formattarli e di gestire opportunamente lo scroll della chat: se la pagina è la prima, viene effettuato uno scroll fino al messaggio più recente, altrimenti viene mantenuta la posizione corrente;
- `showErrorMessage(message: string)`: Si occupa di assegnare l'attributo errorMessage per poter far visualizzare un messaggio di errore all'utente;
- `showTemporaryErrorMessage(message: string, duration: number = 5000)`: Si occupa di assegnare l'attributo errorMessage per poter far visualizzare un messaggio di errore all'utente per un tempo limitato;
- `clearErrorMessage()`: Si occupa di assegnare l'attributo errorMessage ad una stringa vuota per poter nascondere il messaggio di errore all'utente;
- `onLoadMoreMessages()`: Si occupa di gestire il caricamento di ulteriori messaggi dello storico della chat, cioè, se le condizioni sono soddisfatte, aggiorna l'attributo della pagina corrente e chiama il metodo loadOldMessages;
- `onScrollChange(isScrolledUp: boolean)`: Viene assegnato l'attributo showScrollToBottom al valore del parametro isScrolledUp, che indica se l'utente ha scrollato verso l'alto o verso il basso;
- `onSendMessage(text: string)`: Si occupa di aggiungere il messaggio dell'utente alla chat, di chiamare il backend per salvarlo nel database, di chiamare il backend per ottenere la risposta, di aggiungere la risposta alla chat, di chiamare il backend per salvare la risposta nel database e di piazzare a false l'attributo hideSuggestions per ottenere le possibili domande successive;
- `onSuggestionClicked(suggestion: string)`: Si occupa di chiamare il metodo onSendMessage con il parametro suggestion;
- `formatResponse(response: string): string`: Si occupa di formattare la risposta ricevuta dal backend, sostituendo i caratteri Markdown con i corrispondenti tag HTML, e aggiungendo la formattazione per i riquadri dello snippet di codice e dei link correlati;
- `scrollToBottom()`: Si occupa di chiamare il metodo scrollToBottom() del componente ChatMessages per effettuare uno scroll fino al messaggio più recente.

4.2.1.3 ChatMessagesComponent

ChatMessagesComponent rappresenta il componente che si occupa di visualizzare i messaggi della chat. Esso espone i seguenti metodi:

- `ngAfterViewInit()`: Si occupa di associare un addEventListener ai pulsanti di copia di messaggio e snippet di codice, per poter copiare il testo al click;
- `onScroll()`: Si occupa di gestire la barra laterale di scroll associandola alla posizione corrente della chat. Gestisce inoltre i casi di scroll verso il top e verso il bottom della chat: quando avviene uno scorrimento verso il top, viene emesso un segnale per caricare ulteriori messaggi;
- `maintainScrollPosition()`: Si occupa di mantenere la posizione corrente della chat quando vengono caricati nuovi messaggi dallo storico;
- `scrollToBottom()`: Si occupa di effettuare uno scroll in basso fino al messaggio più recente;
- `copyToClipboard(msg: Message)`: Si occupa di copiare il testo del messaggio al click sul pulsante di copia dedicato, formattando opportunamente il testo;
- `copySnippet(code: string, iconElement: HTMLElement)`: Si occupa di copiare lo snippet di codice al click sul pulsante di copia dedicato, formattando opportunamente il codice;
- `stripHtml(html: string): string`: Si occupa di rimuovere i tag HTML dal testo, restituendo una stringa pulita.

4.2.1.4 ChatInputComponent

ChatInputComponent rappresenta il componente che contiene la barra di input e il pulsante Invia. Esso espone il seguente metodo:

- `onSend()`: Viene chiamato quando l'utente clicca sul pulsante Invia oppure preme il tasto Enter. Si occupa, se le condizioni lo consentono, di emettere l'evento di invio del messaggio, passando il testo inserito dall'utente.

4.2.1.5 ChatSuggestionsComponent

ChatSuggestionsComponent rappresenta il componente che contiene le possibili domande successive. Esso espone i seguenti metodi pubblici:

- `ngOnChanges(changes: SimpleChanges)`: Si occupa di ascoltare i cambiamenti all'interno della schermata dei messaggi e di rilevare dunque quando ci sono le condizioni per chiamare il metodo getContinuationSuggestions per il caricamento delle possibili domande successive;
- `onSuggestionClick(text: string)`: Si occupa di emettere l'evento di click su una possibile domanda successiva, passando il testo della domanda.

Inoltre, contiene i seguenti metodi privati:

- `canLoadSuggestions(): boolean`: Si occupa di verificare se ci sono le condizioni per chiamare il metodo getContinuationSuggestions per il caricamento delle possibili domande successive. In particolare, verifica che siano presenti e disponibili i testi dell'ultima domanda e dell'ultima risposta, sulle quali le possibili domande successive si basano;
- `getContinuationSuggestions()`: Si occupa di chiamare il metodo getContinuationSuggestions di ChatService con un payload contenente l'ultima domanda, l'ultima risposta e la quantità di domande successive da generare, per ottenere le possibili domande successive, e le assegna all'attributo continuationSuggestions, che permette di visualizzarle tramite HTML.

4.2.1.6 ChatLoadingIndicatorComponent

ChatLoadingIndicatorComponent rappresenta il componente che visualizza l'indicatore di caricamento durante il caricamento della risposta e dei messaggi dal database. Esso prevede una banana morsicata, cioè il simbolo del gruppo *SWEg Labs*, che gira su se stessa, per indicare all'utente che l'applicazione sta processando. Il componente non espone metodi, in quanto si limita a visualizzare l'indicatore di caricamento.

4.2.1.7 ChatHeaderComponent

ChatHeaderComponent rappresenta il componente che visualizza l'intestazione della chat, contenente il logo di *SWEg Labs* e il titolo dell'applicazione, "BuddyBot". Esso non espone metodi, in quanto si limita a visualizzare l'intestazione della chat. Questo componente contiene il componente ChatBadge, che fa visualizzare il badge di esito dell'ultimo caricamento dei documenti alla destra dell'header.

4.2.1.8 ChatBadgeComponent

ChatBadgeComponent rappresenta il componente che fa visualizzare il badge di esito dell'ultimo caricamento dei documenti. Esso possiede un attributo lastLoadOutcome di tipo LastLoadOutcome, che rappresenta l'esito dell'ultimo caricamento dei documenti nel database vettoriale al momento corrente, che può assumere i valori True, False o Error, i quali sono renderizzati visivamente rispettivamente con una spunta, con una X o con un segnale di pericolo. Viene inoltre presentata una scritta descrittiva accanto al badge, che, in relazione ad esso, assume rispettivamente i seguenti valori: "Aggiornato", "Da aggiornare" ed "Errore". Esso espone i seguenti metodi:

- `ngOnInit()`: Si occupa di inizializzare il componente e di sottoscrivere il proprio attributo lastLoadOutcome al valore dell'attributo lastLoadOutcome$ di DatabaseService;
- `isUpdated(): boolean`: Si occupa di verificare se l'ultimo caricamento dei documenti è stato completato con successo, ritornando true se lastLoadOutcome è True, altrimenti false.

4.2.2 Service

4.2.2.1 ChatService

ChatService rappresenta il servizio che si occupa di gestire la comunicazione tra il frontend e il backend, ovvero di inviare le richieste al backend e di ricevere le risposte. Esso presenta un attributo apiBaseUrl che indica l'indirizzo del backend al quale collegarsi, e un attributo lastMessageTimestamp, che serve per poter associare una data di invio al messaggio inviato al backend. Contiene inoltre i seguenti metodi pubblici:

- `getLastMessageTimestamp(): number`: Si occupa di restituire l'attributo lastMessageTimestamp;
- `setLastMessageTimestamp(time: number)`: Si occupa di assegnare l'attributo lastMessageTimestamp al valore del parametro time;
- `sendMessage(message: string): Observable<{ response: string }>`: Si occupa di inviare il messaggio di tipo stringa al backend, ricevere la risposta di tipo dizionario di stringhe e restituirla;
- `getContinuationSuggestions(payload: {question: string; answer: string; quantity: number;}): Observable<Record<string, string>>`: Si occupa di inviare il payload contenente l'ultima domanda, l'ultima risposta e la quantità di domande successive da generare al backend, ricevere le possibili domande successive e restituirle.

4.2.2.2 DatabaseService

DatabaseService rappresenta il servizio che si occupa di gestire la comunicazione tra il frontend e il backend per quanto riguarda il recupero e salvataggio dei messaggi da e nel database e il recupero dell'esito dell'ultimo caricamento dei documenti nel database vettoriale dal database. Esso presenta un attributo apiBaseUrl che indica l'indirizzo del backend al quale collegarsi, un attributo lastLoadOutcome$ di tipo Observable¡LastLoadOutcome¿, che rappresenta l'esito dell'ultimo caricamento dei documenti nel database vettoriale, e un attributo lastLoadOutcomeSubject di tipo BehaviorSubject¡LastLoadOutcome¿, che serve per poter aggiornare l'attributo lastLoadOutcome$. Contiene inoltre i seguenti metodi pubblici:

- `getMessages(quantity: number, page: number = 1): Observable<Message[]>`: Si occupa di inviare la quantità di messaggi e la pagina da recuperare al backend, ricevere i messaggi come lista di oggetti Message e restituirli;
- `saveMessage(msg: Message): Observable<{ success: boolean; message: string }>`: Si occupa di inviare il messaggio da salvare al backend, ricevere l'esito dell'operazione come dizionario contenente esito del salvataggio e relativo messaggio, e restituirlo;
- `loadLastLoadOutcome()`: Si occupa di inviare una richiesta al backend per recuperare l'esito dell'ultimo caricamento dei documenti nel database vettoriale, ricevere l'esito e aggiornare l'attributo lastLoadOutcomeSubject, che a sua volta viene osservato dall'attributo lastLoadOutcome$.

5 Stato dei Requisiti Funzionali

5.1 Requisiti funzionali

Codice	Rilevanza	Descrizione	Stato
ROF1	Obbligatorio	L'utente deve poter inserire un'interrogazione in linguaggio naturale nel sistema.	✓
ROF2	Obbligatorio	Quando l'interrogazione viene invata al sistema, deve essere generata una risposta.	✓
ROF3	Obbligatorio	Nel caso il sistema fallisca nel generare una risposta per via di un problema interno, deve far visualizzare all'utente un messaggio di errore, chiedendo di riprovare più tardi.	✓
ROF4	Obbligatorio	Nel caso in cui l'utente inserisca un'interrogazione che non riguarda i contenuti del database associato, il sistema deve rispondere all'utente che la domanda inserita è fuori contesto.	✓
ROF5	Obbligatorio	Nel caso il sistema non riesca a trovare le informazioni richieste dall'utente nonostante siano correlate al contesto, deve rispondere all'utente spiegando la mancanza dell'informazione richiesta.	✓
RDF6	Desiderabile	L'utente deve poter visualizzare i link dei file da cui il sistema ha preso i dati per la risposta.	✓
RDF7	Desiderabile	Nel caso non sia possibile recuperare i link dei file utilizzati per generare la risposta, deve essere visualizzato un messaggio di errore.	✓
RZF8	Opzionale	Deve essere presente un pulsante al cui click la risposta del chatbot viene copiata nel dispositivo dell'utente.	✓
RDF9	Desiderabile	Nel caso la risposta contenga uno snippet di codice, deve essere presente un pulsante che permetta di copiare il singolo snippet nel dispositivo dell'utente.	✓
RDF10	Desiderabile	Deve essere presente un sistema di archiviazione delle domande e delle risposte in un database relazionale.	✓
RDF11	Desiderabile	L'utente deve poter visualizzare lo storico dei messaggi, recuperato dal database relazionale.	✓
RDF12	Desiderabile	Nel caso il sistema fallisca nel recuperare lo storico della chat, deve essere fatto visualizzare un messaggio di errore all'utente spiegando che non è stato possibile recuperare lo storico.	✓
ROF13	Obbligatorio	Uno scheduler deve collegarsi al sistema e periodicamente aggiornare il database vettoriale con i dati più recenti.	✓
ROF14	Obbligatorio	La risposta deve essere generata prendendo in considerazione i dati di contesto provenienti da GitHub, Jira e Confluence	✓
ROF15	Obbligatorio	Il sistema deve poter convertire i documenti ottenuti in formato vettoriale.	✓

Codice	Rilevanza	Descrizione	Stato
ROF16	Obbligatorio	Il sistema deve poter aggiornare il database vettoriale con i nuovi documenti ottenuti.	✓
RZF17	Opzionale	Se la conversazione non è ancora avviata l'utente deve poter visualizzare e selezionare alcune domande di partenza proposte.	✗
RZF18	Opzionale	Dopo la visualizzazione di una risposta, all'utente devono venire suggerite alcune interrogazioni che è possibile porre al sistema per proseguire la conversazione.	✓
RZF19	Opzionale	Nel caso il sistema vada in errore nel tentativo di proporre alcune interrogazioni per proseguire la conversazione, deve venire mostrato un messaggio che comunica l'errore all'utente e invita a fare altre domande.	✓
RZF20	Opzionale	Il sistema deve mostrare a schermo un badge sopra alla schermata della chat.	✓
RZF21	Opzionale	Il sistema deve comunicare all'utente l'esito dell'ultimo tentativo di aggiornamento del database vettoriale utilizzando un badge.	✓
ROF22	Obbligatorio	Il sistema deve processare i documenti che vengono caricati, creandone i loro embedding.	✓
ROF23	Obbligatorio	Il sistema deve salvare, in modo persistente, il contenuto dei documenti caricati.	✓
ROF24	Obbligatorio	Il sistema deve salvare, in modo persistente, i metadati dei documenti caricati.	✓
ROF25	Obbligatorio	Il sistema deve salvare, i commit e i file scritti in caratteri testuali di GitHub, escludendo PDF o immagini.	✓
ROF26	Obbligatorio	Il sistema deve permettere all'utente di digitare una domanda libera in linguaggio naturale tramite una barra di input.	✓
RZF27	Opzionale	L'utente deve poter selezionare una delle domande proposte dal sistema, e inviarla come interrogazione al chatbot.	✓
RDF28	Desiderabile	Il sistema deve mostrare a schermo i messaggi dell'utente presenti nello storico.	✓
RDF29	Desiderabile	Il sistema deve mostrare a schermo i messaggi del chatbot presenti nello storico.	✓
RDF30	Desiderabile	I messaggi del chatbot e dell'utente mostrati a schermo devono essere distinguibili tra loro tramite una colorazione differente.	✓
RDF31	Desiderabile	Il sistema deve caricare e mostrare a schermo i messaggi cronologicamente precedenti quando l'utente, tramite scroll verso il basso raggiunge l'inizio della lista dei messaggi mostrati.	✓

Codice	Rilevanza	Descrizione	Stato
RDF32	Desiderabile	Nel caso il sistema fallisca nel recupeare i messaggi precedenti dal database, l'utente deve visualizzare un messaggio che comunica che non è stato possibile recuperare altri messaggi dal database.	✓
RDF33	Desiderabile	Nel caso non ci siano altri messaggi nel database, l'utente deve visualizzare un messaggio che comunica che non ci sono altri messaggi da visualizzare.	✗
RZF34	Opzionale	Il sistema deve comunicare all'utente se l'ultimo tentativo di aggiornamento del database vettoriale ha avuto successo.	✓
RZF35	Opzionale	Il sistema deve comunicare all'utente se l'ultimo tentativo di aggiornamento del database vettoriale è fallito.	✓
RZF36	Opzionale	Il sistema deve comunicare all'utente se non è stato possibile recuperare l'esito dell'ultimo aggiornamento del database vettoriale.	✓
RZF37	Opzionale	L'utente, accanto al badge che segnala l'esito dell'aggiornamento automatico del database vettoriale, deve visualizzare un messaggio che spiega il significato di quest'ultimo nella sua forma corrente.	✓
RZF38	Opzionale	Se l'ultimo tentativo di aggiornamento del database vettoriale ha avuto successo, l'utente accanto al badge deve visualizzare un messaggio che spiega il significato della forma a spunta di quest'ultimo.	✓
RZF39	Opzionale	Se l'ultimo tentativo di aggiornamento del database vettoriale è fallito, l'utente accanto al badge deve visualizzare un messaggio che spiega il significato della forma a X di quest'ultimo.	✓
RZF40	Opzionale	Se non è stato possibile recuperare l'esito dell'ultimo tentativo di aggiornamento del database vettoriale, l'utente accanto al badge deve visualizzare un messaggio che spiega il significato della forma a segnale di pericolo di quest'ultimo.	✓
RZF41	Opzionale	Ad ogni tentativo di aggiornamento del database vettoriale il sistema deve salvare in un file di testo un log che contenga le seguenti informazioni: fonte dei dati (GitHub, Jira o Confluence), esito, orario.	✓
RZF42	Opzionale	Ad ogni tentativo di aggiornamento del database vettoriale il sistema deve salvare in un file di testo un log che contenga le seguenti informazioni: fonte dei dati (GitHub, Jira o Confluence), esito, orario, numero di file aggiunti, numero di file modificati, numero di file eliminati.	✓
RDF43	Desiderabile	Il sistema deve mostrare a schermo la data e l'ora di ogni messaggio presente nello storico.	✓

Codice	Rilevanza	Descrizione	Stato
RDF44	Desiderabile	Nel caso non sia possibile recuperare la data e l'ora di un messaggio presente nello storico, deve essere visualizzato al suo posto un avviso di errore.	✓
RZF45	Opzionale	Il sistema deve mostrare un riquadro di caricamento durante la generazione di una risposta.	✓
RZF46	Opzionale	Il sistema deve mostrare un riquadro di caricamento durante il caricamento dello storico dei messaggi.	✓
RZF47	Opzionale	Il sistema deve mostrare un riquadro di caricamento durante il caricamento dei messaggi precedenti.	✓
RZF48	Opzionale	Deve essere presente un pulsante che, al click, permetta all'utente di tornare ai messaggi più recenti della chat.	✓

5.2 Requisiti soddisfatti

Tutti i 14 requisiti funzionali obbligatori sono stati soddisfatti.
Dei requisiti funzionali desiderabili ne sono stati soddisfatti 13 su 14: in particolare, l'RDF33 non è stato soddisfatto perchè, in accordo con il *proponente*$_G$, è stato valutato che un messaggio che comunica che non ci sono altri messaggi da visualizzare nello storico è ridondante, e potrebbe essere scambiato per un messaggio di errore.
Infine, dei requisiti funzionali opzionali ne sono stati soddisfatti 19 su 20: in particolare, l'RZF17 non è stato soddisfatto perchè è stata notata un'incongruenza logica dello stesso con RDF28 ed RDF29, confermata anche dal *proponente*: infatti, il caricamento dei messaggi presenti nello storico comporta che, dal secondo accesso all'applicazione in poi, l'utente possa visualizzare tutti i messaggi precedenti, ed allora non esiste più un "inizio della conversazione" in occasione del quale sia possibile proporre delle domande iniziali.