Università di Padova

Corso di Ingegneria del Software

Specifica architetturale

Versione 1.0

Storia del documento

Versione	Data	Autori	Verificatori	Descrizione
1.0	2024/04/03	Andrea Cecchin		Approvazione documento
0.20	2024/04/03	Francesco Giacomuzzo	Leonardo Lago	Stesura sezione 4.3.
0.19	2024/04/03	Andrea Cecchin	Marco Dolzan	Stesura sezione 4.5.7.
0.18	2024/03/31	Andrea Cecchin	Francesco Ferraioli	Aggiornamento sezione 4.5.
0.17	2024/03/30	Marco Dolzan	Francesco Giacomuzzo	Aggiornamento capitolo 3.
0.16	2024/03/30	Andrea Cecchin	Francesco Giacomuzzo	Aggiornamento sezione 4.6.
0.15	2024/03/30	Francesco Giacomuzzo	Andrea Cecchin	Inizio stesura sezione 4.6.
0.14	2024/03/28	Anna Nordio	Marco Dolzan	Aggiornamento sezione 4.5.5.
0.13	2024/03/28	Francesco Giacomuzzo	Francesco Ferraioli	Aggiornamento sezione 4.4.
0.12	2024/03/28	Andrea Cecchin	Francesco Ferraioli	Aggiornamento sezione 4.4.
0.11	2024/03/28	Francesco Ferraioli	Andrea Cecchin	Stesura sezione 4.5.6.

Versione	Data	Autori	Verificatori	Descrizione
0.10	2024/03/26	Giovanni Menon	Leonardo Lago	Stesura sezione 4.5.4.
0.9	2024/03/24	Leonardo Lago	Andrea Cecchin	Stesura sezione 4.5.3.
0.8	2024/03/23	Marco Dolzan	Leonardo Lago	Stesura sezione 4.5.2.
0.7	2024/03/23	Marco Dolzan	Francesco Giacomuzzo	Stesura sezione 4.5.1.
0.6	2024/03/21	Francesco Giacomuzzo	Andrea Cecchin	Stesura sezione 4.5.
0.5	2024/03/21	Giovanni Menon	Andrea Cecchin	Stesura sezione 4.2.
0.4	2024/03/16	Andrea Cecchin	Francesco Giacomuzzo	Inizio stesura sezione 4.4.
0.3	2024/03/10	Francesco Ferraioli	Andrea Cecchin	Stesura sezione 4.1.
0.2	2024/02/26	Andrea Cecchin	Anna Nordio	Stesura sezione 3.
0.1	2024/02/10	Andrea Cecchin	Anna Nordio	Creazione documento, stesura capitoli 1 e 2.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

1.1 Scopo del documento

Il documento ha l'obiettivo di descrivere dettagliatamente l'architettura logica e di deployment, i design *pattern*$_G$ adottati e le tecnologie impiegate nella realizzazione del prodotto per il *progetto*$_G$ *Knowledge Management*$_G$ *AI*$_G$.

1.2 Glossario

Al fine di prevenire ed evitare possibili ambiguità nei termini e acronimi presenti all'interno della documentazione, è stato realizzato un glossario dove sono riportati i relativi significati (vedasi Glossario_v2.0). All'interno di ogni documento i termini specifici, che quindi hanno una definizione all'interno del Glossario, saranno contrassegnati con una 'G' aggiunta a pedice e trascritti in corsivo. Tale prassi sarà rispettata solamente per la prima occorrenza del termine o acronimo.

1.3 Riferimenti

1.3.1 Normativi

- Norme_di_progetto_v2.0;
- *Capitolato*$_G$ d'appalto C1: `https://www.math.unipd.it/~tullio/IS-1/2023/Progetto/C1.pdf` *(Ultimo accesso: 2024/04/03)*;
- Verbali esterni;
- Regolamento Progetto Didattico: `https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/PD2.pdf` *(Ultimo accesso: 2024/04/03)*.

1.3.2 Informativi

- Analisi_dei_requisiti_v3.0;

- Piano_di_qualifica_v2.0;
- Slide dell'insegnamento di Ingegneria del Software, in particolare:
  - I pattern architetturali:
    `https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf` *(Ultimo accesso: 2024/04/03)*;
  - Il pattern *Dependency Injection*$_G$:
    `https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Architetturali%20-%20Dependency%20Injection.pdf` *(Ultimo accesso: 2024/04/03)*;
  - Progettazione Software:
    `https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/T6.pdf` *(Ultimo accesso: 2024/04/03)*;
  - Qualità del Software:
    `https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/T7.pdf` *(Ultimo accesso: 2024/04/03)*;
  - Pattern della GoF:
    `https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf` *(Ultimo accesso: 2024/04/03)*;
    `https://www.math.unipd.it/~rcardin/swea/2021/Design%20Pattern%20Comportamentali_4x4.pdf` *(Ultimo accesso: 2024/04/03)*;
    `https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Strutturali.pdf` *(Ultimo accesso: 2024/04/03)*;
  - Programmazione SOLID:
    `https://www.math.unipd.it/~rcardin/swea/2021/SOLID%20Principles%20of%20Object-Oriented%20Design_4x4.pdf` *(Ultimo accesso: 2024/04/03)*.
- Sito ufficiale *ChromaDB*$_G$:
  `https://www.trychroma.com` *(Ultimo accesso: 2024/04/03)*.
- Documentazione ufficiale *LangChain*$_G$ JS:
  `https://js.langchain.com/docs/get_started/introduction` *(Ultimo accesso: 2024/04/03)*.
- Documentazione ufficiale *MinIO*$_G$:
  `https://min.io/docs/minio/kubernetes/upstream/` *(Ultimo accesso: 2024/04/03)*.
- Documentazione ufficiale Next.js:
  `https://nextjs.org/docs` *(Ultimo accesso: 2024/04/03)*.

- Sito ufficiale Node.js:
  `https://nodejs.org/en` *(Ultimo accesso: 2024/04/03)*.
- Sito ufficiale *OpenAI*$_G$ *API*$_G$:
  `https://openai.com/blog/openai-api` *(Ultimo accesso: 2024/04/03)*.
- Sito ufficiale *React*$_G$:
  `https://it.legacy.reactjs.org` *(Ultimo accesso: 2024/04/03)*.
- Appunti su "*Clean*$_G$ Architecture" di Robert C. Martin:
  `https://mirkorap16.gitbook.io/clean-architecture/architettura-clean` *(Ultimo accesso: 2024/04/03)*.

Descrizione del prodotto

2.1 Scopo del prodotto

Lo scopo del prodotto è fornire alle aziende un'applicazione di knowledge management che permetta ai propri lavoratori di ricevere risposte, chiare e immediate, a domande inerenti al contesto lavorativo.
Questa applicazione deve permettere a ogni dipendente di ottenere informazioni sul funzionamento dei macchinari aziendali, sui processi di lavorazione dei diversi prodotti, sulle norme di sicurezza da rispettare, o più in generale istruzioni su come eseguire una azione di qualsiasi tipo, pur rimanendo vincolati al contesto d'uso.
Le informazioni date all'utente, tramite un'interfaccia $chatbot_G$, sono contenute nei documenti inseriti nel sistema dall'azienda stessa, e dovranno essere esposte in modo chiaro e comprensibile ad ogni lavoratore, così da rendere le informazioni utili più accessibili e velocemente reperibili.

2.2 Funzionalità

Questa applicazione prevede due principali funzionalità: la gestione della documentazione aziendale, e l'interazione con un'interfaccia chat per reperire le informazioni contenute nei documenti.
La gestione dei documenti deve permettere il caricamento di un nuovo documento, l'eliminazione e la consultazione dei documenti già presenti.
Nell'interfaccia del chatbot, l'utente deve essere in grado di fare una domanda riguardante un'informazione contenuta all'interno dei documenti aziendali, e ricevere una risposta corretta, chiara ed esaustiva.

Tecnologie

Nella seguente sezione del documento, vengono elencate le principali tecnologie utilizzate nella realizzazione del prodotto. Esse sono corredate da una descrizione generale, dalla versione di riferimento, dalla descrizione del loro scopo e funzionamento nel prodotto, e i motivi che hanno portato alla loro scelta.

3.1 ChromaDB

Descrizione: Base di dati che permette l'archiviazione e la persistenza di vettori.

Versione: 1.8.1

Utilizzo: È stato utilizzato per memorizzare i vettori di *embeddings*$_G$, generati a partire dal contenuto di documenti testuali.

Motivazione: Uno tra i vector *database*$_G$ più diffusi ed utilizzati, è facilmente integrabile con LangChain, altra tecnologia utilizzata. A differenza di alternative come Pinecone, oltre ad essere completamente gratuito ed a permettere la creazione e gestione di un numero illimitato di collezioni di vettori, rende possibile associare dei metadati ai vettori archiviati.

3.2 Docker

Descrizione: Piattaforma di sviluppo e gestione di applicazioni che permette di creare, distribuire e eseguire in software in container virtualizzati.

Versione: 24.0.7

Utilizzo: Utilizzato nell'installazione del prodotto software.

Motivazione: Utilizzando un container con tutte le dipendenze tecnologiche definite e soddisfatte, permette una più semplice preparazione e utilizzo del prodotto.

3.3 Jest

Descrizione: *Framework*$_G$ di testing per codice *Javascript*$_G$ e Typescript.

Versione: 29.1.2

Utilizzo: È stato utilizzato nell'implementazione ed esecuzione dei test di unità e integrazione.

Motivazione: È la tecnologia più utilizzata su Javascript e Typescript per testare il codice. Inoltre la sua grande popolarità semplifica la ricerca di documentazioni e informazioni per poter utilizzare tale tecnologia al meglio.

3.4 LangChain

Descrizione: Framework per lo sviluppo di applicazioni LLM_G-based.

Versione: 0.1.24

Utilizzo: Viene utilizzato per il parsing e lo splitting dei documenti testuali, oltre che per la generazione degli embeddings e il processo di information retrieval.

Motivazione: Oltre a permettere la gestione di moltissime funzionalità diverse tramite l'utilizzo di chain, offre l'integrazione con un'ampia varietà di altre tecnologie.

3.5 MinIO

Descrizione: Object store.

Versione: 8.4.3

Utilizzo: Utilizzato per lo store dei documenti presenti nel sistema.

Motivazione: È un object store gratuito, ad elevate prestazioni e completamente compatibile con il servizio di storage offerto da Amazon S3. Rispetto alla conservazione dei documenti nel file system, questa soluzione offre un maggiore livello di sicurezza offerta dagli elevati ed ottimizzati metodi di crittografia utilizzati durante il caricamento dei dati.

3.6 NextJs

Descrizione: Framework per lo sviluppo di applicazioni web in React.

Versione: 14.1.0

Utilizzo: È stato utilizzato per lo sviluppo del prodotto.

Motivazione: Offre soluzioni standard per la gestione di aspetti comuni, come il routing. Inoltre, permette di scegliere, tra diversi tipi, quale runtime utilizzare.

3.7 NodeJs

Descrizione: Runtime system per esecuzione di codice Javascript.

Versione: 20.11.0

Utilizzo: È utilizzato per l'esecuzione del prodotto.

Motivazione: Principale sistema di esecuzione di codice Javascript e Typescript per popolarità e prestazioni.

3.8 Ollama

Descrizione: Framework per l'esecuzione di LLM locali.

Versione: 0.1.19

Utilizzo: È stato utilizzato per l'esecuzione in locale del LLM utilizzato nell'embedding dei documenti e per il funzionamento del chatbot integrato.

Motivazione: Oltre ad essere completamente gratuito, Ollama offre una vastissima gamma, in continuo aggiornamento, di modelli *open source*$_G$ tra cui scegliere.

3.9 OpenAI

Descrizione: Fornitore di servizi legati all'intelligenza artificiale.

Versione: -

Utilizzo: Tramite le proprie API, viene utilizzato il LLM GPT-3.5 per l'embedding dei documenti e per il funzionamento del chatbot integrato.

Motivazione: Leader indiscusso del settore, offre servizi a pagamento con le migliori prestazioni. La sua notorietà nel settore garantisce il suo supporto a praticamente qualsiasi tecnologia inerente all'ambito AI.

3.10 Postgres

Descrizione: Sistema di database relazionale open source.

Versione: 8.11.3

Utilizzo: È utilizzato per la memorizzazione delle informazioni relative alle sessioni di conversazione e ai messaggi in essi scambiati.

Motivazione: Rispetto ad altri database, come *SQLite*$_G$, permette la gestione di una mole di dati elevata, coerentemente al reale caso d'uso del prodotto. Più in generale, assicura una elevata scalabilità e buone performance.

3.11 React

Descrizione: Libreria Javascript per lo sviluppo di applicazioni web e interfacce utente.

Versione: 18.2.0

Utilizzo: È stato utilizzato per lo sviluppo del prodotto.

Motivazione: Rispetto alla principale alternativa *Angular*$_G$ ha una curva di apprendimento ridotta, che comporta una più semplice interazione.

3.12 Shadcn/ui

Descrizione: Libreria di *componenti*$_G$ React.

Versione: 0.8.0

Utilizzo: È stata utilizzata per facilitare la codifica del *front-end*$_G$, incorporando nell'interfaccia grafica componenti prefabbricate, personalizzabili e altamente riutilizzabili.

Motivazione: Oltre ad essere una libreria popolare è molto utilizzata, è stata adottata perché permette l'utilizzo di componenti accessibili, personalizzabili, riutilizzabili e altamente modularizzate. Quest'ultimo aspetto ci permette di ottenere un front-end in linea con il Compounds Component pattern, tipico di React.

3.13 Tailwind

Descrizione: Framework CSS_G per lo sviluppo semplificato della presentazione di siti e applicazioni web.

Versione: 3.3.0

Utilizzo: Viene utilizzato per sviluppare l'aspetto dell'interfaccia del prodotto.

Motivazione: Utilizzando delle classi standard preesistenti, permette la definizione semplificata e più rapida del codice CSS.

3.14 Tsyringe

Descrizione: Libreria per la Dependency Injection con Typescript.

Versione: 4.8.0

Utilizzo: Questa tecnologia è utilizzata per registrare in container, gestire e risolvere le dipendenze delle classi, realizzando l'Inversion of Control.

Motivazione: La registrazione e risoluzione delle dipendenze è estremamente semplice ed intuitiva grazie all'uso dei decorators supportati da Typescript.

3.15 Typescript

Descrizione: Linguaggio di programmazione, estensione di Javascript.

Versione: 5.3.0

Utilizzo: Linguaggio utilizzato per lo sviluppo del prodotto.

Motivazione: Grazie all'introduzione della tipizzazione statica rispetto a Javascript, Typescript permette il rilevamento di errori a tempo di compilazione.

3.16 Riepilogo

Nome tecnologia	Versione
ChromaDB	1.8.1
Docker	24.0.7
Jest	29.1.2
LangChain	0.1.24
MinIO	8.4.3
NextJs	14.1.0
NodeJs	20.11.0
Ollama	0.1.19
OpenAI	-
Postgres	8.11.3
React	18.2.0
Shadcn/ui	0.8.0
Tailwind	3.3.0
Tsyringe	4.8.0
Typescript	5.3.0

Tabella 1: Tecnologie utilizzate

Architettura

4.1 Logica del prodotto

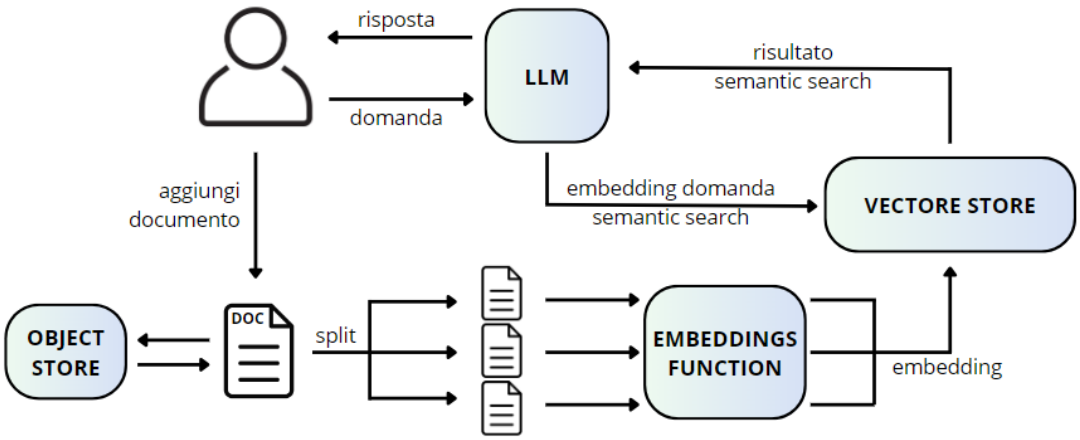

Figura 1: Funzionamento del prodotto

Le funzionalità offerte dall'applicazione si suddividono in due categorie: la gestione dei documenti presenti nel sistema, con la possibilità per l'utente di aggiungerne e rimuoverne, e la loro consultazione tramite chatbot.
Nella gestione della documentazione presente e utilizzata dal sistema, l'utente può effettuare il caricamento dei documenti dei quali vorrà ricevere le informazioni. I documenti verranno quindi salvati e archiviati tramite MinIO, e successivamente processati. In particolare, ogni documento viene diviso nelle sue singole pagine, e ogni pagina viene suddivisa in chunk, i quali sono successivamente embeddizzati, ovvero elaborati e trasformati in vettori di numeri reali. I valori dello spazio vettoriale generato rappresentano così le parole che compongono i vari documenti. Questo avviene sfruttando funzionalità di embedding date dai large language model utilizzati nel prodotto, ovvero un modello locale di Ollama e GPT-3.5 di OpenAI, e i vettori sono successivamente salvati in modo persistente all'interno del vector store ChromaDB. Qui sono storicizzati in attesa di essere utilizzati in seguito per la formazione delle risposte.
Nell'interazione con i documenti presenti nel sistema, l'utente potrà inviare delle do-

mande al chatbot e ricevere risposte riguardanti i documenti caricati in precedenza. Nella formazione e invio della domanda posta dall'utente, il sistema effettua la vettorizzazione della domanda tramite lo stesso processo di embedding utilizzato per i documenti. Per fornire una risposta esatta all'utente, viene effettuata una semantic search sul database vettoriale, andando a comparare la distanza dello spazio vettoriale della domanda, con quello delle informazioni contenute all'interno del vectore store.
Una volta completata la semantic search, e identificata la risposta alla domanda posta grazie alla vicinanza del vettore della domanda con quello della risposta, il sistema fornisce il risultato della ricerca semantica al modello large language, che si occuperà della generazione della risposta per l'utente. Nel caso in cui la domanda dell'utente non riguardi i documenti caricati, o più in generale quando non è presente la risposta alla domanda in nessuno dei documenti presenti nel sistema, viene fornita una risposta standard, che informa la non pertinenza della domanda. Questo è possibile grazie a operazioni di prompt engineering nella formulazione della richiesta inoltrata al large language model.
Nel caso in cui la risposta alla domanda sia stata identificata in un documento, il chatbot fornisce la risposta, riportando anche il documento che è fonte della risposta data, mostrandone nome e numero della pagina di interesse.
Durante la conversazione tra utente e chatbot, i messaggi scambiati sono salvati in modo persistente sul database Postgres, così da poter ricomporre la chat history quando l'utente esce dalla pagina del chatbot. I messaggi vengono eliminati solo quando è l'utente a richiederne l'eliminazione.
Oltre a poter inserire nuovi documenti nel sistema, l'utente potrà rimuovere e visionare quelli già presenti e memorizzati su MinIO. Alla rimozione di un documento, sono contestualmente eliminati anche i vettori degli embeddings delle sue pagine. In questo modo, il chatbot non potrà più restituire risposte sul documento non più nel sistema.

4.2 Architettura di sistema

L'architettura adottata nella realizzazione dell'applicativo segue le linee guida date dalla Clean Architecture. Il software è stato suddiviso in livelli, in base alla tipologia di logica ad esso riferita, secondo una struttura concentrica.
Ogni livello o strato dell'applicativo è indipendente da quelli ad esso più esterni, ovvero nulla presente in un livello più interno può conoscere l'implementazione del livello più esterno.
Questo principio alla base dell'architettura clean ci permette di ottenere un prodotto software la cui logica di business è del tutto indipendente dal framework utilizzato per la realizzazione dell'interfaccia grafica e dalle basi di dati adottate.
Gli elementi che costituiscono l'architettura sono i seguenti:

Controller: è il punto di ingresso per le richieste dall'esterno del sistema. Esso accetta le richieste in arrivo dall'esterno, utilizza gli use case per effettuare le operazioni richieste e gestisce la conversione dei dati ricevuti nei formati appropriati.

Use case: rappresentano le singole funzionalità o casi di utilizzo dell'applicazione. Essi contengono la logica di business. Essi coordinano l'esecuzione delle operazioni richieste dalla logica di business, andando a definire ed implementare le regole di business specifiche di ogni funzionalità. Utilizzano i repository per accedere e manipolare i dati.

Repository: sono responsabili per l'accesso ai dati persistenti, fornendo un'interfaccia astratta per la comunicazione con il livello di persistenza dei dati sottostante. Astraggono il resto dell'applicazione dai dettagli tecnici specifici del sistema di persistenza dei dati, fornendo i metodi per recuperare, salvare, aggiornare ed eliminare dati.

Data source: sono gli elementi che effettivamente accedono e manipolano i dati nel sistema di persistenza sottostante, forniscono l'implementazione concreta dei metodi per l'accesso ai dati presenti nei repository.

L'adozione di una Clean Architecture come architettura di sistema porta numerosi vantaggi:

- Elevato grado di modularità del codice, suddividendolo in base alla logica e alle responsabilità che implementa.
- Indipendenza della business logic da qualsiasi framework, tecnologie e interfaccia utente adottata. Ogni cambiamento apportato non compromette il corretto funzionamento del sistema.

- Il funzionamento del sistema è completamente indipendente dalle tecnologie utilizzate per i database, con le fonti dei dati che possono essere molteplici. Ogni cambiamento apportato non compromette il corretto funzionamento del sistema.
- La leggibilità, la comprensibilità e la testabilità dell'intero sistema è semplificata.

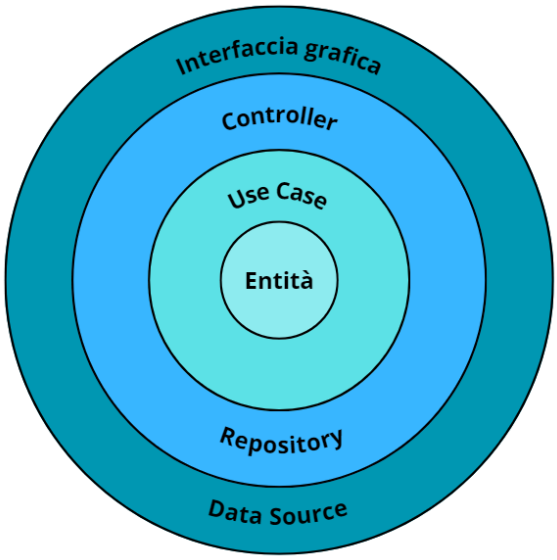

Figura 2: Schema della Clean Architecture

4.3 Architettura di deployment

Nel valutare quale architettura di deployment adottare per l'applicativo, è stato considerato il caso d'utilizzo realistico del prodotto.
Essendo l'applicazione pensata per essere utilizzata all'interno di fabbriche o piccole aziende, senza particolari necessità di espansioni del sistema o modifiche sostanziali una volta installato il prodotto, è stato ritenuto vantaggioso e utile optare per un'architettura monolite. Oltre ad essere la scelta migliore per lo scopo finale e le caratteristiche del prodotto, permettendo anche una più semplice fase di progettazione, codifica e test, il monolite non presenta le stesse difficoltà che avrebbe portato la scelta di altre architetture di deployment, come quella a micro-servizi, la quale sarebbe stata di difficile implementazione considerate anche le competenze e le conoscenze dei membri del gruppo.

Il deploy del prodotto avviene tramite containerizzazione con Docker Compose. In questo modo, è possibile facilitare l'installazione dell'applicativo andando a definire un ambiente dove tutte le dipendenze sono risolte, nel quale sono già predisposti correttamente tutti i servizi necessari al corretto utilizzo del prodotto. In particolare, il container predispone al corretto utilizzo dei database, ovvero ChromaDB, Postgres e MinIO, oltre ai servizi di Ollama.

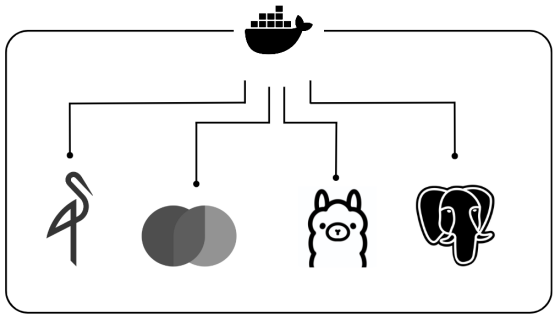

Figura 3: Servizi predisposti dal Docker Compose

4.4 Architettura Front-End

L'intera applicazione è stata sviluppata in React. La scelta di tale tecnologia ha portato l'interfaccia grafica ad essere sviluppata in componenti, composti tra di loro fino a configurare la GUI finale.
Per ricercare una modularità del codice, così da aumentare la leggibilità e la manutenibilità dello stesso, sono stati creati più componenti per ogni pagina, che interagiscono e si compongono tra di loro, fino a costituire la pagina finale. Questo approccio, tipico di React e dei pattern adottati per il front-end, permette di suddividere il codice, evitando che esso venga inserito interamente nella singola pagina.
L'interfaccia utente prevede due pagine, definite dalle due macro funzionalità dell'applicazione. Una pagina è dedicata alla gestione della documentazione, l'altra all'interazione con il chatbot.
Gli elementi comuni alle due pagine principali sono stati implementati con uguali componenti, così da riutilizzare il codice ove possibile, senza dover ridefinire gli stessi elementi grafici più volte.

Nella descrizione della architettura che costituisce il front-end dell'applicazione, sarà riportato un UML delle componenti principali di ogni pagina e per ognuna di esse sarà riportata una descrizione dettagliata, con una lista degli elementi che la compongono e dei requisiti associati a tale componente.

4.4.1 Documents Page

La Documents Page è la pagina relativa alla gestione della documentazione presente nell'applicazione. È possibile visualizzare quali sono i documenti che l'utente ha già inserito, le loro informazione, la form per effettuare l'upload di un nuovo documento e i bottoni per visualizzarli ed eliminarli.
È presente una sidebar a comparsa, comune a quello presente anche nella Chat Page, dove è posizionato il form per il caricamento dei documenti e dove è possibile modificare le impostazioni dell'applicazione.

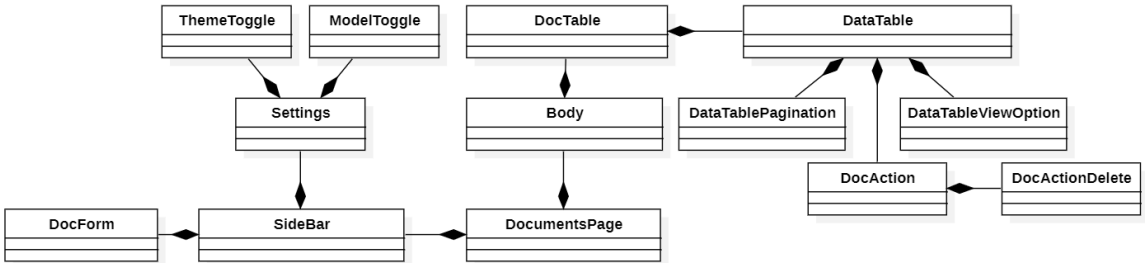

Figura 4: UML dei componenti principali della Documents Page

SideBar

Descrizione
Questo componente costituisce il menù a tendina comune alle due pagine dell'applicazione. In esso è presente il componente Settings, dove tramite ModelToggle e ThemeToggle è possibile cambiare il LLM e il tema utilizzato dall'applicazione.
Oltre alle impostazioni generali dell'applicativo, in questo componente viene posizionato il DocForm utilizzato per caricare nuovi componenti nel sistema.

Elementi
Il valore booleano isCollapsed è gestito dallo stato del componente, e determina se mostrare o meno il menù a comparsa. La modifica di questo valore è associato all'evento onClick di un bottone.
Il componente Settings, nel quale sono inseriti i due componenti Toggle, è costituito da un Dialog che mostra il suo contenuto mediante l'azione di un DialogTrigger. In questo modo, ModelToggle e ThemeToggle sono renderizzati solo a seguito dell'interazione con il pulsante del menù.
All'interno di questi due componenti Toggle, viene impostato il tema e il modello da utilizzare modificando un valore controllato dallo stato dei componenti stessi.

Requisiti associati

- RFO-1;
- RFO-2.

Body

Descrizione
Il componente Body rappresenta il corpo della pagina. In esso sono inseriti tutti i componenti necessari alla visualizzazione dei documenti dell'applicazione, contenuti da DocTable.

Elementi
Al suo interno è contenuto il solo componente DocTable.

DocForm

Descrizione
Conenuto all'interno della Sidebar, DocForm è il componente che contiene la form con cui effettuare l'upload di nuovi documenti nel sistema. Qui grazie ad un Input è possibile selezionare il documento di interesse ed inviarlo grazie ad un apposito bottone.

Elementi
Grazie ad un componente Input che permette la selezione di soli file con estensione .pdf, .docx e .mp3, è possibile inviare un documento grazie ad un Button, a cui è associata la funzione handleFormSubmit tramite evento onClick. All'interno di questa funzione è verificato il valore di selectedFile, il quale viene utilizzato per fornire i dati necessari per effettuare le effettive operazioni per caricare il documento.
Per accertare il corretto formato del documento selezionato, viene utilizzata la funzione handleFileChange, invocata ogni qual volta viene utilizzato il form Input, che deseleziona ogni file con estensione diversa da quella aspettata, mostrando contestualmente un messaggio d'errore.

Requisiti associati

- RFO-29;
- RFO-30;
- RFO-31;
- RFD-32;
- RFD-34;
- RFO-66;
- RFD-67;

- RFD-68;
- RFZ-69.

DocTable

Descrizione
La DocTable è il componente che definisce la tabella nella quale sono visualizzati i documenti presenti nell'applicazione.

Elementi
Tramite l'utilizzo dello hook useEffect, ogni volta che viene cambiato il modello utilizzato dall'applicazione vengono recuperate le informazioni sui documenti presenti a sistema. Questi dati, definiti dall'interfaccia DocumentInfo e composti dal nome del documento (string), dalla data di inserimento a sistema (string), dalla dimensione (number) e dai tag assegnati (string), sono utilizzati per formare il componente DataTable che, costruendo la tabella, è responsabile di renderizzare la lista finale dei documenti.
Il valore booleano isLoading identifica se è in corso o meno l'operazione di recupero dei documenti. Esso viene utilizzato per renderizzare, internamente a DataTable, il componente IsLoadingDoc, che mostra un componente Skeleton momentaneo.

DataTable

Descrizione
È il componente responsabile della visualizzazione delle informazioni dei documenti. Oltre alla tabella, contiene i componenti responsabili alla ricerca, al filtraggio, alla visualizzazione e all'eliminazione dei documenti, oltre che a DataTablePagination e DataTableViewOption. Queste ultime sono utilizzate per gestire la logica di renderizzazione della tabella.

Elementi
I dati che costituiscono la tabella sono inseriti all'interno di componenti adibiti a questo preciso ruolo, quali Table, TableBody, TableCell, TableHead, TableHeader e TableRow.
La tabella contiene in ogni sua riga le informazioni dei documenti presenti, oltre che le azioni sul documento (visualizzazione, eliminazione e cambio tag) tramite il componente DocAction.
La ricerca, effettuata tramite la digitazione su un componente Input, permette di mostrare i documenti in base al loro nome e data di inserimento, tramite una ricerca che agisce sulle string di nome e data, tramite un valore setFilterValue. La ricerca avviene per data (true) o per nome (false) in base al valore booleano della variabile

setIsFilterData, che varia con un evento onClick di due componenti DropdownMenuItem.
I componenti DataTablePagination e DataTableViewOption, posti all'interno di DataTable, agiscono sulla paginazione della tabella.

Requisiti associati

- RFO-3;
- RFO-4;
- RFO-5;
- RFO-6;
- RFZ-7;
- RFD-9;
- RFO-10;
- RFO-11;
- RFO-12;
- RFO-13.

DataTablePagination e DataTableViewOption

Descrizione
È il componente che gestisce la logica di renderizzazione della tabella, dividendo i dati della stessa in più pagine e permettendone la navigazione mediante pulsanti.

Elementi
Tramite i componenti Select, SelectContent, SelectItem, SelectTrigger e SelectValue, DataTablePagination divide i dati presenti in tabella, così da mostrarne per pagina tanti quanti il valore di pageSize. Questo valore è stabilito dall'interazione con un componente Select, che agisce all'evento onValueChange chiamando setPageSize, funzione responsabile della modifica effettiva del valore numerico.
La navigazione nella tabella avviene con l'evento onClick di vari Button, che tramite le funzioni setPageIndex, previousPage e nextPage variano i dati da mostrare a schermo.
Nel componente DataTableViewOption, tramite un DropdownMenu contenente dei DropdownMenuCheckboxItem, è presente la logica che permette la renderizzazione delle sole colonne della tabella selezionate.

DocAction

Descrizione
La componente DocAction contiene tutti gli elementi necessari ad effettuare le azioni

previste sui documenti, ovvero visualizzazione, eliminazione e cambio tag.

Elementi
Grazie ad un menù a comparsa formato con i componenti DropdownMenu, DropdownMenuContent, DropdownMenuItem, DropdownMenuLabel, DropdownMenuSeparator e DropdownMenuTrigger, DocAction permette la visualizzazone del documento di interesse grazie all'evento onClick del DropdownMenuItem. Questo evento chiama la funzione handleShowDoc che, recuperato dal back-end l'url del documento, apre in modalità blank una nuova scheda del browser, dove poter visualizzare il documento. L'azione di eliminazione è invece lasciata al componente DocActionDelete, contenuto da DocAction.

Requisiti associati

- RFO-8;
- RFZ-15;
- RFZ-16;
- RFD-36;
- RFD-37.

DocActionDelete

Descrizione
È la componente responsabile della logica con cui l'utente può eliminare un documento.

Elementi
Dall'interazione con un AlertDialogTrigger, viene aperto un AlertDialog per confermare l'eliminazione del documento dal sistema. All'evento onClick dell'AlertDialogAction, viene chiamata la funzione handleDelteDoc con cui viene richiesta al back-end l'eliminazione del documento e di ogni informazioni ad esso associata.

Requisiti associati

- RFO-27;
- RFO-28.

4.4.2 Chat Page

La Chat Page è la pagina relativa allo scambio di messaggi tra l'utente ed il chatbot integrato nell'applicazione. Nella schermata principale, è possibile visualizzare i messaggi inviati dall'utente, le risposte del chatbot, le fonti delle risposte e l'orario in cui sono stati inviati i messaggi.
È presente una sidebar laterale, nella quale sono presenti dei bottoni che permettono di visualizzare informazioni aggiuntive riguardanti le chat avviate in precedenza, un buttone per creare nuove chat, uno per eliminarle e un tasto con le impostazioni generali dell'applicazione, comune anche alla Documents Page.

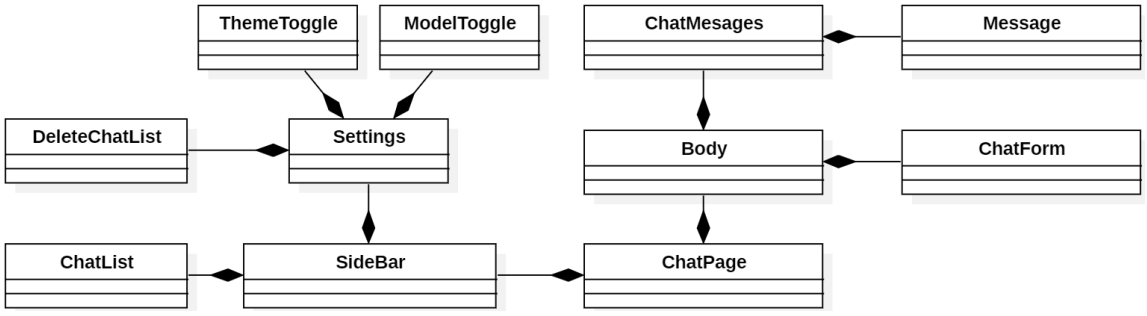

Figura 5: UML dei componenti principali della Chat Page

SideBar

Descrizione
Questo componente costituisce il menù a tendina comune alle due pagine dell'applicazione. In esso è presente il componente Settings, dove tramite ModelToggle e ThemeToggle è possibile cambiare il LLM e il tema utilizzato dall'applicazione. Con DeleteChatList è invece possibile eliminare tutte le conversazioni ancora attive.
Inoltre, tramite la ChatList, è possibile visualizzare la lista delle sessioni di chat effettuate e attive nel sistema, oltre ad un bottone per creare nuove chat.

Elementi
La struttura della SideBar è identica a quella della DocumentPage. L'unica differenza riguarda i componenti presenti al suo interno.
Il componente Settings contenuto in SideBar, oltre ad aver ThemeToggle e ModelToggle, presenta DeleteChatList, adibito alla eliminazione di tutte le sessioni del sistema.
La lista delle sessioni è renderizzata tramite il componente ChatList, contenuto nella SideBar, che prende il posto del DocForm presente all'interno della Documents Page.

Requisiti associati

- RFO-1;
- RFO-2.

Body

Descrizione
Il componente Body rappresenta il corpo della pagina. In esso sono inseriti tutti i componenti necessari alla visualizzazione dell'area di chat, contenuti da ChatForm e ChatMessages.

Elementi
Al suo interno sono presenti i componenti ChatForm e ChatMessages.

ChatList

Descrizione
Il componente ChatList è la parte dell'interfaccia utente responsabile della gestione e visualizzazione delle sessioni di chat, consentendo agli utenti di selezionare, creare ed eliminare sessioni di chat, e fornendo un feedback visivo durante il caricamento dei dati.

Elementi
Il componente ChatList utilizza l'hook useChatsData per gestire i dati delle chat e fornisce funzionalità per la creazione e l'eliminazione di chat. L'elenco delle chat è mostrato all'interno di un'area scrollabile, con opzioni per eliminare le chat esistenti. Quando viene eliminata una chat, lo stato viene aggiornato per riflettere la modifica. ChatList è composta da una serie di elementi Button, i quali rappresentano le chat avviate in precedenza nell'applicazione. Tramite funzioni asincrone, se i bottoni vengono cliccati allora verranno evidenziati, andando ad indicare quale chat sta venendo visualizzata nella sezione centrale della pagina. Per ciascun bottone, è inserito lateralmente un ulteriore bottone, il quale una volta cliccato, fa apparire un DropDownMenu, composto da DropDownMenuTrigger, DropDownMenuContent, DropDownMenuLabel, DropDownMenuSeparator. La voce del menù è un ulteriore bottone che permette la cancellazione della singola chat.

Requisiti associati

- RFD-48;
- RFD-49;
- RFO-53;
- RFO-54.

DeleteChatList

Descrizione
Il componente DeleteChatList fornisce un'interfaccia utente per eliminare tutte le sessioni di chat all'interno dell'applicazione, inclusa una finestra di dialogo di conferma, la gestione delle azioni e l'aggiornamento della sezione laterale dei threads al momento della cancellazione riuscita.

Elementi
Il componente utilizza l'hook useChatsData per gestire lo stato delle chat e il componente AlertDialog per confermare l'azione di eliminazione. Quando l'utente conferma l'eliminazione, la funzione handleDeleteAllChat viene chiamata per eseguire l'operazione e aggiornare lo stato della chat. In caso di errore, viene visualizzato un messaggio.

Requisiti associati

- RFD-50;
- RFD-51.

ChatForm

Descrizione
Questa componente è responsabile della raccolta dell'input dell'utente per i messaggi di chat, della validazione dell'input, della gestione dei dati della sessione di chat e dell'invio dei messaggi al server.

Elementi
Il componente ChatForm utilizza l'hook useForm di react-hook-form per gestire lo stato e la validazione del form, insieme a due altri hook per gestire i dati dei messaggi e delle chat. Il form include un campo di testo per l'input del messaggio e un pulsante di invio. Quando l'utente invia il messaggio, il componente gestisce la validazione, l'invio dei dati e la visualizzazione di eventuali errori tramite un messaggio di errore.

Requisiti associati

- RFO-40;
- RFO-41.

ChatMessages e Message

Descrizione
Il componente ChatMessages è responsabile della visualizzazione dei messaggi all'in-

terno di una sessione di chat, differenziando tra i messaggi generati dall'utente e quelli generati dall'intelligenza artificiale, gestendo la visualizzazione dei messaggi quando non ce ne sono, e fornendo una Scroll Area per una navigazione agevole attraverso la cronologia della chat.

Elementi
Sfruttando l'hook useMessagesData, il componente gestisce i dati dei messaggi e utilizza il componente Message per rendere ogni messaggio visibile. La distinzione tra i messaggi generati dall'AI e quelli dell'utente è chiaramente identificata, con una rappresentazione visiva diversa per ciascun tipo di messaggio. Inoltre, offre un'interfaccia utente intuitiva per la visualizzazione e la navigazione dei messaggi all'interno della chat.

Requisiti associati

- RFO-44;
- RFO-47;
- RFO-52;
- RFO-55;
- RFO-56;
- RFD-57.

4.5 Architettura Back-End

Coerentemente alla struttura imposta dall'adozione di una Clean Architecture, il back-end dell'applicazione è organizzato tramite delle classi controller, use case, repository e data source.
Per collegare l'interfaccia grafica lato front-end con il funzionamento dell'applicativo lato back-end, sono state implementate delle server actions per chiamare i controller relativi all'azione desiderata.
I controller eseguono la chiamata allo use case relativo, che costituisce ed implementa la business logic dell'intero sistema.
Per accedere alle fonti dei dati, necessari a realizzare le funzionalità definite dai casi d'uso, le classi use cases si relazionano con delle classi repositories, nelle quali è posta la logica di persistenza, che presentano i metodi per lettura, scrittura ed eliminazione dei dati.
La singola classe repository si interfaccia alla classe data source ad essa associata, la quale presenta i metodi che intervengono concretamente sulle fonti dei dati per compiere le funzioni richieste sui database.
Una volta terminate le operazioni nel data source, il controllo torna alla repository, la quale ritorna i dati recuperati allo use case quando richiesto. Completata la funzionalità definita dal caso d'uso, il flusso di controllo torna al controller che dovrà gestire la risposta verso la server action da cui tutto è iniziato.

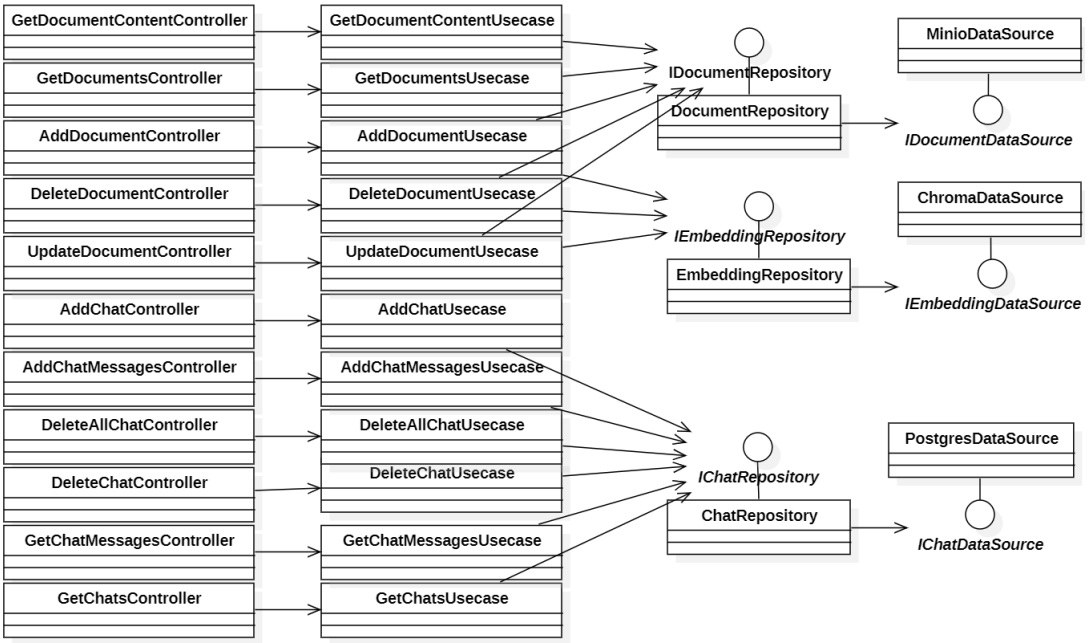

Figura 6: UML introduttivo delle classi del back-end

4.5.1 Server Actions

addDocument

Parametri

- *data: FormData.*

Descrizione
Questa funzione asincrona lato server effettua una chiamata a AddDocumentController, passando il parametro data. Data contiene una *string* model e un *File* file, che rappresentano il modello che deve presentare il nuovo documento e il file da aggiungere. Gestisce l'aggiunta di un documento inviando i dati al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

deleteDocument

Parametri

- *name: string*;
- *model: IModel.*

Descrizione
Questa funzione asincrona lato server effettua una chiamata a DeleteDocumentController, passando i parametri name e model, che rappresentano il nome del documento da eliminare e il modello che non deve più presentare tale documento. Gestisce l'eliminazione di un documento inviando i dati al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

getDocument

Parametri

- *model: IModel.*

Descrizione
Questa funzione asincrona lato server effettua una chiamata a GetDocumentsController, passando il parametro model, che rappresenta il modello da cui prendere i documenti associati. Gestisce il recupero delle informazioni dei documenti inviando i dati al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

getDocumentContent

Parametri

- *docName: string*;
- *model: IModel*.

Descrizione
Questa funzione asincrona lato server effettua una chiamata a GetDocumentContentController, passando i parametri docName e model, che rappresentano il nome del documento da cui recuperare il link per la visualizzazione e il modello che presenta tale documento. Gestisce il recupero dell'informazione inviando i dati al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

updateDocument

Parametri

- *docName: string*;
- *model: IModel*;
- *updatedMetadas: Metadatas*.

Descrizione
Questa funzione asincrona lato server effettua una chiamata a UpdateDocumentController, passando i parametri docName, model e updatedMetadas, che rappresentano il nome del documento da aggiornare, il modello che presenta tale documento e il nuovo valore che deve avere il tag di visibilità del documento. Gestisce la modifica delle informazioni inviando i dati al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

addChat

Parametri

- *title: string*.

Descrizione
Questa funzione asincrona lato server effettua una chiamata a AddChatController, passando il parametro title, che rappresenta il nome della sessione di conversazione da aggiungere. Gestisce la creazione della sessione inviando i dati al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

addChatMessages

Parametri

- *messages: ICustomMessages.*

Descrizione
Questa funzione asincrona lato server effettua una chiamata a AddChatMessagesController, passando il parametro messages, che rappresenta il messaggio da salvare relativo ad una sessione di conversazione. Gestisce il salvataggio del messaggio inviando il dato al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

deleteAllChat

Descrizione
Questa funzione asincrona lato server effettua una chiamata a DeleteAllChatController per eliminare tutti i dati relativi a sessioni e chat history. Gestisce l'eliminazione di queste informazioni effettuando una chiamata al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

deleteChat

Parametri

- *id: number.*

Descrizione
Questa funzione asincrona lato server effettua una chiamata a DeleteChatController, passando il parametro id, che rappresenta il valore univoco della sessione di conversazione da eliminare. Gestisce l'eliminazione della sessione inviando i dati al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

getChatMessages

Parametri

- *id: number.*

Descrizione
Questa funzione asincrona lato server effettua una chiamata a GetChatMessagesController, passando il parametro id, che rappresenta il valore univoco della sessione di conversazione da cui recuperare la chat history. Gestisce il recupero dati della sessione inviando il parametro al controller e si occupa di gestire eventuali errori che si possono verificare durante il processo.

getChats

Descrizione
Questa funzione asincrona lato server effettua una chiamata a GetChatsController, con cui recupera le informazioni delle sessioni di conversazione attive nell'applicazione. Gestisce il recupero dati delle sessioni ed eventuali errori che si possono verificare durante il processo.

4.5.2 Controllers

AddDocumentController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: AddDocumentUsecase*.

Metodi

- *handle(data: FormData): Promise<Response>*.

Descrizione
Questa classe controller, chiamata tramite la server action addDocument, gestisce la chiamata al AddDocumentUsecase per aggiungere a sistema un nuovo documento, passando un *FormData* contenente una *string* model e un *File* file, che rappresentano il modello che deve presentare il nuovo documento e il file da aggiungere.

Dipendenze

- AddDocumentUsecase.

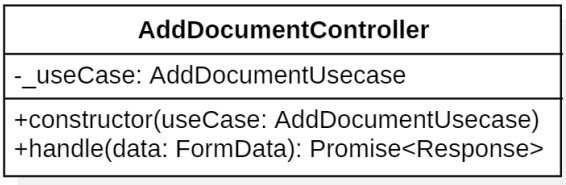

Figura 7: UML della classe AddDocumentController

DeleteDocumentController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: DeleteDocumentUsecase*.

Metodi

- *handle(docName: string, model: IModel): Promise<Response>.*

Descrizione

Questa classe controller, chiamata tramite la server action deleteDocument, gestisce la chiamata al DeleteDocumentUsecase per eliminare dal sistema un documento, passando una *string* docName e una *string* model, che rappresentano il nome del documento da eliminare e il modello da cui rimuoverlo.

Dipendenze

- DeleteDocumentUsecase.

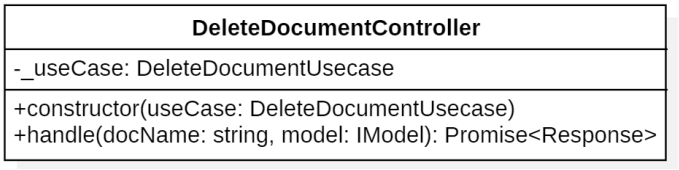

Figura 8: UML della classe DeleteDocumentController

UpdateDocumentController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: UpdateDocumentUsecase.*

Metodi

- *handle(docName: string, model: IModel, updatedMetadas: Metadatas): Promise<Response>.*

Descrizione

Questa classe controller, chiamata tramite la server action updateDocument, gestisce la chiamata al UpdateDocumentUsecase per aggiornare i tag e i metadati associati ad un documento, passando una *string* docName, una *string* model e dei metadati updatedMetadas, che rappresentano il nome del documento da aggiornare, il modello di appartenenza e il nuovo valore del tag di visibilità associato.

Dipendenze

- UpdateDocumentUsecase.

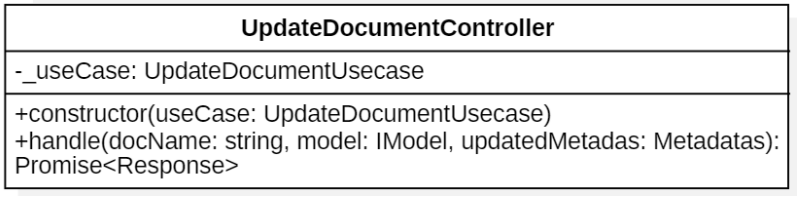

Figura 9: UML della classe UpdateDocumentController

GetDocumentContentController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: GetDocumentContentUsecase*.

Metodi

- *handle(docName: string, model: IModel): Promise<Response>*.

Descrizione

Questa classe controller, chiamata tramite la server action getDocumentContent, gestisce la chiamata al GetDocumentContentUsecase per recuperare dal sistema il link di visualizzazione di un documento, passando una *string* docName e un *IModel* (stringa con il nome di un modello) model, che rappresentano il nome del documento di interesse e il modello che presenta il documento.

Dipendenze

- GetDocumentContentUsecase.

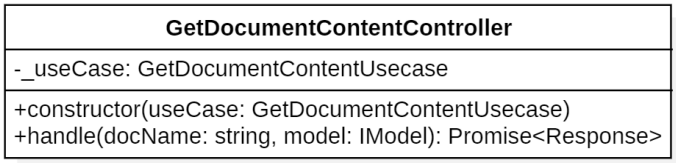

Figura 10: UML della classe GetDocumentContentController

GetDocumentsController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: GetDocumentsUsecase*.

Metodi

- *handle(model: IModel): Promise<Response>*.

Descrizione
Questa classe controller, chiamata tramite la server action getDocuments, gestisce la chiamata al GetDocumentsUsecase per recuperare le informazioni dei documenti quali nome, data di ultima modifica e dimensione, passando una *string* model, che rappresenta il nome del modello da cui recuperare i documenti.

Dipendenze

- GetDocumentsUsecase.

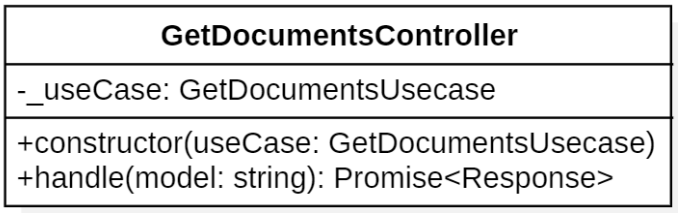

Figura 11: UML della classe GetDocumentsController

AddChatController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: AddChatUsecase*.

Metodi

- *handle(title: string): Promise<Response>.*

Descrizione
Questa classe controller, chiamata tramite la server action addChat, gestisce la chiamata al AddChatUsecase per creare una nuova sessione di conversazione, passando una *string* title che rappresenta il nome che prenderà quella sessione.

Dipendenze

- AddChatUsecase.

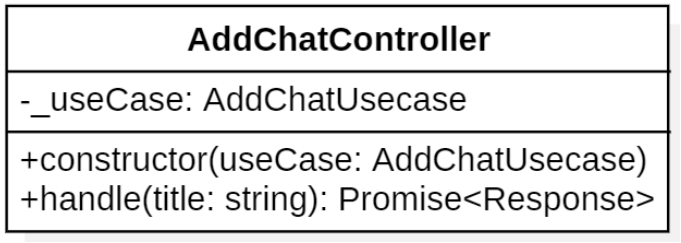

Figura 12: UML della classe AddChatController

AddChatMessagesController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: AddChatMessagesUsecase.*

Metodi

- *handle(messages: ICustomMessages): Promise<Response>.*

Descrizione
Questa classe controller, chiamata tramite la server action addChatMessages, gestisce la chiamata al AddChatMessagesUsecase dove viene passato un nuovo messaggio scambiato in una sessione (messages: *ICustomMessages*) da aggiungere nel database.

Dipendenze

- AddChatMesagesUsecase.

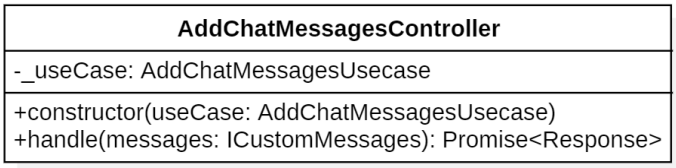

Figura 13: UML della classe AddChatMessagesController

DeleteAllChatController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: DeleteAllChatUsecase*.

Metodi

- *handle(): Promise<Response>*.

Descrizione

Questa classe controller, chiamata tramite la server action deleteAllChat, gestisce la chiamata al DeleteAllChatUsecase per rimuovere nel database tutte le sessioni e le chat history relative.

Dipendenze

- DeleteAllChatUsecase.

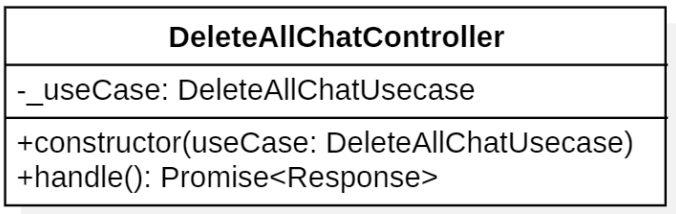

Figura 14: UML della classe DeleteAllChatController

DeleteChatController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: DeleteChatUsecase*.

Metodi

- *handle(id: number): Promise<Response>*.

Descrizione

Questa classe controller, chiamata tramite la server action deleteChat, gestisce la chiamata al DeleteChatUsecase per rimuovere dal database una particolare sessione, identificata dal *number* id, e la chat history relativa.

Dipendenze

- DeleteChatUsecase.

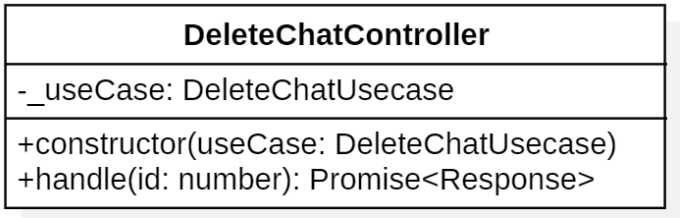

Figura 15: UML della classe DeleteChatController

GetChatMessagesController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: GetChatMessagesUsecase*.

Metodi

- *handle(id: number): Promise<Response>*.

Descrizione
Questa classe controller, chiamata tramite la server action getChatMessages, gestisce la chiamata al GetChatMessagesUsecase per recuperare dal database la chat history di una particolare sessione, identificata dal *number* id.

Dipendenze

- GetChatMessagesUsecase.

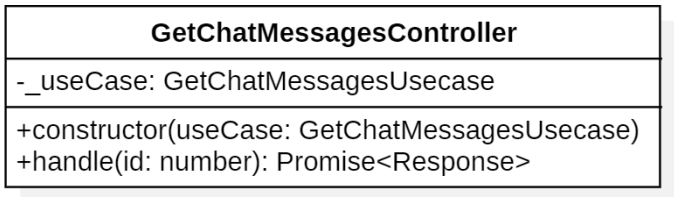

Figura 16: UML della classe GetChatMessagesController

GetChatsController

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_useCase: GetChatsUsecase*.

Metodi

- *handle(): Promise<Response>*.

Descrizione
Questa classe controller, chiamata tramite la server action getChats, gestisce la chiamata al GetChatsUsecase per recuperare dal database le sessioni di conversazione attive nel sistema.

Dipendenze

- GetChatsUsecase.

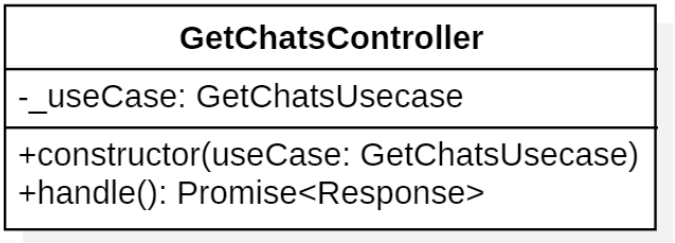

Figura 17: UML della classe GetChatsController

4.5.3 Use Cases

AddDocumentUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_documentRepository: IDocumentRepository*;
- private readonly *_embeddingRepository: IEmbeddingRepository*;
- private readonly *_embeddingsFunction*.

Metodi

- *execute(file: File, model: IModel)*.

Descrizione
Questo use case, chiamato dal AddDocumentController, contiene la business logic relativa alle operazioni per aggiungere un nuovo documento nel sistema. Ricevuto il file e la *string* model, gestisce la chiamata al DocumentRepository per aggiungere il nuovo documento. Dopo aver effettuato il parsing del contenuto del documento e averlo diviso per pagine, viene effettuato l'embedding del testo utilizzando l'embedding function determinata dal modello desiderato. Una volta completato questo processo, mapando gli embedding con il nome del documento, i metadati associati e il suo file, viene chiamato anche l'EmbeddingRepository per lo storage dei vettori sul database vettoriale.
Presenta un riferimento a delle interfacce IDocumentRepository e IEmbeddingRepository, le cui implementazioni concrete sono DocumentRepository e EmbeddingRepository, che espongono i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IDocumentRepository;

- IEmbeddingRepository.

Requisiti associati

- RFO-29;
- RIO-1;
- RIO-2;
- RIO-3;
- RIO-4.

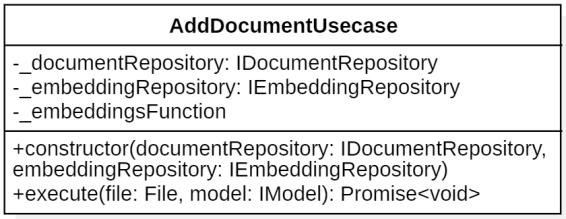

Figura 18: UML della classe AddDocumentUsecase

DeleteDocumentUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_documentRepository: IDocumentRepository*;
- private readonly *_embeddingRepository: IEmbeddingRepository*.

Metodi

- *execute(docName: string, model: IModel).*

Descrizione
Questo use case, chiamato dal DeleteDocumentController, contiene la business logic relativa alle operazioni per eliminare uno specifico documento dal sistema. Ricevute le *string* docName e model, gestisce la chiamata al DocumentRepository e all'EmbeddingRepository per rimuovere nuovo documento.
Dopo aver effettuato la chiamata al DocumentRepository per richiedere l'eliminazione del documento dal database MinIO, richiede l'id degli embedding da rimuovere all'EmbeddingRepository, per poi richiedere l'eliminazione dei vettori identificati da essi alla medesima classe repository.

Presenta un riferimento a delle interfacce IDocumentRepository e IEmbeddingRepository, le cui implementazioni concrete sono DocumentRepository e EmbeddingRepository, che espongono i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IDocumentRepository;
- IEmbeddingRepository.

Requisiti associati

- RFO-27.

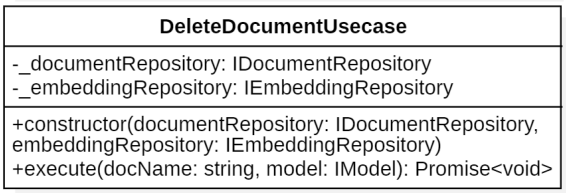

Figura 19: UML della classe DeleteDocumentUsecase

UpdateDocumentUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_documentRepository: IDocumentRepository*;
- private readonly *_embeddingRepository: IEmbeddingRepository*.

Metodi

- *execute(docName: string, model: IModel, updatedMetadas: Metadatas)*.

Descrizione

Questo use case, chiamato dal UpdateDocumentController, contiene la business logic relativa alle operazioni per aggiornare i tag e i metadati, relativi alla visibilità, di uno specifico documento. Ricevute le *string* docName e model e i nuovi metadati updatedMetadas, gestisce la chiamata al DocumentRepository e all'EmbeddingRepository per aggiornare i valori.

Dopo aver effettuato la chiamata al DocumentRepository per richiedere l'aggiornamento del tag di visibilità del documento nel database MinIO, richiede l'id degli embedding da modificare all'EmbeddingRepository, per poi richiedere la modifica del metadato visibility dei vettori identificati da essi alla medesima classe repository. Presenta un riferimento a delle interfacce IDocumentRepository e IEmbeddingRepository, le cui implementazioni concrete sono DocumentRepository e EmbeddingRepository, che espongono i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IDocumentRepository;
- IEmbeddingRepository.

Requisiti associati

- RFZ-15;
- RFZ-16;
- RFD-36;
- RFD-37.

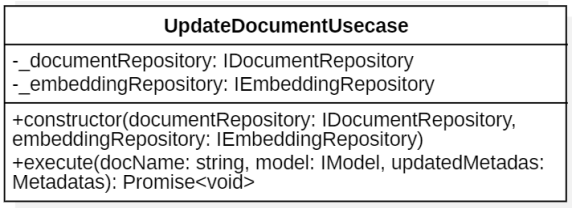

Figura 20: UML della classe UpdateDocumentUsecase

GetDocumentContentUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_documentRepository : IDocumentRepository*.

Metodi

- *execute(docName: string, model: IModel)*.

Descrizione
Questo use case, chiamato dal GetDocumentContentController, contiene la business logic relativa alle operazioni per ottenere il link di visualizzazione di uno specifico documento dal sistema. Ricevute le *string* docName e model, gestisce la chiamata al DocumentRepository per recuperare tale informazione.
Presenta un riferimento ad un'interfaccia IDocumentRepository , la cui implementazione concreta è DocumentRepository, che espone i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IDocumentRepository.

Requisiti associati

- RFO-8;
- RFO-55.

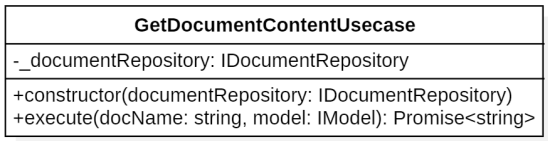

Figura 21: UML della classe GetDocumentContentUsecase

GetDocumentsUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_documentRepository : IDocumentRepository*.

Metodi

- *execute(model: IModel) : Promise<Document[]>*.

Descrizione
Questo use case, chiamato dal GetDocumentsController, contiene la business logic relativa alle operazioni per ottenere le informazioni dei documenti presenti nel sistema. Ricevuto il model, gestisce la chiamata al DocumentRepository per recuperare

le informazioni dei documenti presenti in quello specifico modello.
Presenta un riferimento ad un'interfaccia IDocumentRepository , la cui implementazione concreta è DocumentRepository, che espone i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IDocumentRepository.

Requisiti associati

- RFO-4;
- RFO-5;
- RFO-6;
- RFZ-7;
- RFD-9;
- RFO-10.

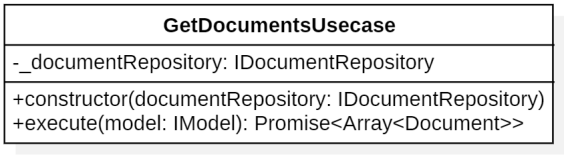

Figura 22: UML della classe GetDocumentsUsecase

AddChatMessagesUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_chatRepository: IChatRepository.*

Metodi

- *execute(messages: ICustomMessages): Promise<void>.*

Descrizione
Questo use case, chiamato dal AddChatMessagesController, contiene la business logic relativa alle operazioni per aggiungere le informazioni dei messaggi scritti, definiti dall'*ICustomMessages*. Ricevuto il messages, gestisce la chiamata al ChatRepository per richiedere il salvataggio del dato.
Presenta un riferimento ad un'interfaccia IChatRepository , la cui implementazione concreta è ChatRepository, che espone i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IChatRepository.

Requisiti associati

- RFO-52.

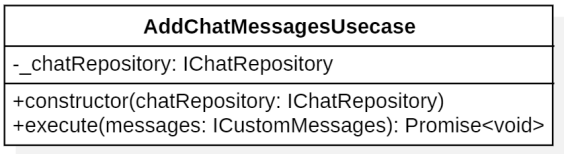

Figura 23: UML della classe AddChatMessagesUsecase

AddChatUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_chatRepository: IChatRepository*.

Metodi

- *execute(title: string): Promise<void>*.

Descrizione
Questo use case, chiamato dal AddChatController, contiene la business logic relativa alle operazioni per aggiungere una nuova sessione di conversazione nel sistema, che avrà un nome definito dalla *string* title. Ricevuto questo title, gestisce la chiamata al ChatRepository per richiedere la creazione della nuova sessione.

Presenta un riferimento ad un'interfaccia IChatRepository , la cui implementazione concreta è ChatRepository, che espone i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IChatRepository.

Requisiti associati

- RFD-48.

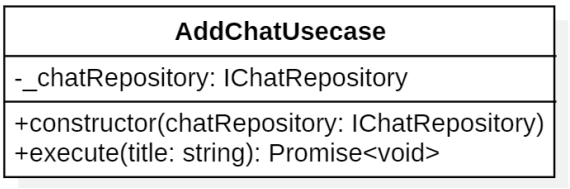

Figura 24: UML della classe AddChatUsecase

DeleteAllChatUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_chatRepository: IChatRepository.*

Metodi

- *execute(): Promise<number>.*

Descrizione
Questo use case, chiamato dal DeleteAllChatController, contiene la business logic relativa alle operazioni per eliminare dal database tutte le informazioni relative alle sessioni di conversazione attive, incluse le chat history ad esse associate.
Presenta un riferimento ad un'interfaccia IChatRepository , la cui implementazione concreta è ChatRepository, che espone i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IChatRepository.

Requisiti associati

- RFD-50;
- RFD-51.

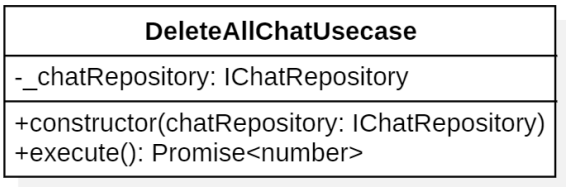

Figura 25: UML della classe DeleteAllChatUsecase

DeleteChatUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_chatRepository: IChatRepository*.

Metodi

- *execute(id: number): Promise<void>*.

Descrizione
Questo use case, chiamato dal DeleteChatController, contiene la business logic relativa alle operazioni per eliminare dal database tutte le informazioni relative ad una particolare sessione di conversazione, definita dal *number* id, e alla sua chat history. Ricevuto l'id, gestisce la chiamata al ChatRepository per richiedere l'eliminazione della sessione.
Presenta un riferimento ad un'interfaccia IChatRepository , la cui implementazione concreta è ChatRepository, che espone i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IChatRepository.

Requisiti associati

- RFD-53;
- RFD-54.

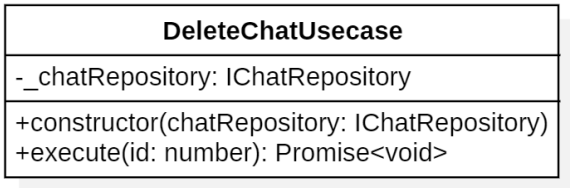

Figura 26: UML della classe DeleteChatUsecase

GetChatMessagesUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_chatRepository: IChatRepository*.

Metodi

- *execute(id: number): Promise<allMessages: Message[], source: MessageSource>*.

Descrizione

Questo use case, chiamato dal GetChatMessagesController, contiene la business logic relativa alle operazioni per ottenere dal database la chat history relativa ad una particolare sessione di conversazione, definita dal *number* id. Ricevuto l'id, gestisce la chiamata al ChatRepository per il recupero dei messaggi.
Presenta un riferimento ad un'interfaccia IChatRepository , la cui implementazione concreta è ChatRepository, che espone i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IChatRepository.

Requisiti associati

- RFO-52.

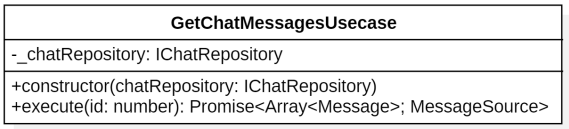

Figura 27: UML della classe GetChatMessagesUsecase

GetChatsUsecase

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_chatRepository: IChatRepository*.

Metodi

- *execute(): Promise<Chat[]>*.

Descrizione

Questo use case, chiamato dal GetChatsController, contiene la business logic relativa alle operazioni per ottenere dal database le informazioni relative alle sessioni di conversazione salvate e attive nel sistema.
Presenta un riferimento ad un'interfaccia IChatRepository , la cui implementazione concreta è ChatRepository, che espone i metodi per lavorare sulle fonti dei dati presenti nel database.

Dipendenze

- IChatRepository.

Requisiti associati

- RFD-49.

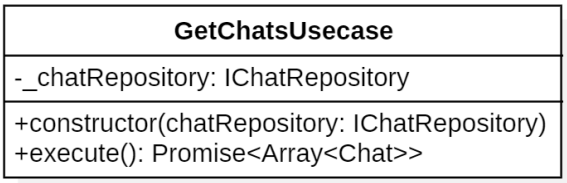

Figura 28: UML della classe GetChatsUsecase

4.5.4 Repositories

DocumentRepository

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private *_documentDataSource: IDocumentDataSource*.

Metodi

- *addDocument(doc: Document, model: IModel): Promise<void>*;
- *deleteDocument(docName: string, model: IModel): Promise<void>*;
- *updateDocument(docName: string, model: IModel, tag: Metadatas): Promise<void>*;
- *getDocuments(model: IModel): Promise<Document[]>*;
- *getDocumentContent(docName: string, model: IModel): Promise<string>*.

Descrizione

La classe DocumentRepository, utilizzata dagli use cases AddDocument, DeleteDocument, UpdateDocument, GetDocumentContent e GetDocuments, espone i metodi con cui interfacciarsi alle fonti dei dati per il recupero dei dati necessari alla logica di business per aggiunta, rimozione, modifica e lettura dei documenti.
Presenta un riferimento ad un'interfaccia IDocumentDataSource, la cui implementazione concreta è MinioDataSource che espone i metodi per lavorare sui dati presenti nel database.

Dipendenze

- IDocumentDataSource.

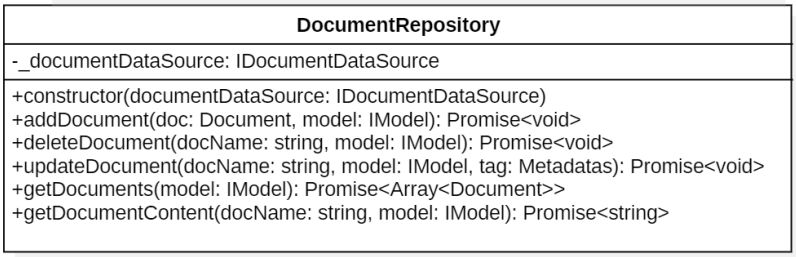

Figura 29: UML della classe DocumentRepository

EmbeddingRepository

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private *_embeddingsDataSource: IEmbeddingDataSource*.

Metodi

- *addEmbedding(embeddings: Embeddings, model: IModel): Promise<void>*;
- *deleteEmbedding(ids: string[], model: IModel): Promise<void>*;
- *updateMetadatas(metadatas: Metadatas, model: IModel, ids: string[]): Promise<void>*;
- *getIdsEmbedding(docName: string, model: IModel): Promise<any>*.

Descrizione

La classe EmbeddingRepository, utilizzata dagli use cases AddDocument e DeleteDocument, espone i metodi con cui interfacciarsi alle fonti dei dati per il recupero dei dati necessari all'aggiunta ed eliminazione degli embedding di documenti.
Presenta un riferimento ad un'interfaccia IEmbeddingDataSource, la cui implementazione concreta è ChromaDataSource che espone i metodi per lavorare sui dati presenti nel vector database.

Dipendenze

- IEmbeddingDataSource.

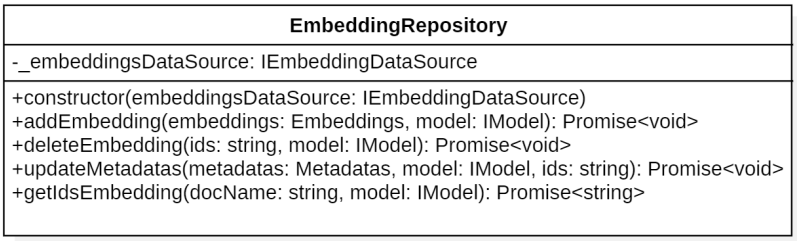

Figura 30: UML della classe EmbeddingRepository

ChatRepository

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private *_chatDataSource: IChatDataSource*.

Metodi

- *addChat(title: string): Promise<number>*;
- *addChatMessages(messages: ICustomMessages): Promise<void>*;
- *deleteAllChat(): Promise<void>*;
- *deleteChat(id: number): Promise<void>*;
- *getChatMessages(id: number): Promise<allMessages: Message[], source: MessageSource>*;
- *getChats(): Promise<Chat[]>*.

Descrizione

La classe ChatRepository, utilizzata dagli use cases AddChatMessages, AddChat, DeleteAllChat, DeleteChat, GetChatMessages e GetChats, espone i metodi con cui interfacciarsi alla fonte dei dati per il recupero delle informazioni necessarie alla logica di business per aggiunta, rimozione e recupero delle sessioni di conversazione e dei messaggi delle chat history ad esse associate.
Presenta un riferimento ad un'interfaccia IChatDataSource, la cui implementazione concreta è PostgresDataSource che espone i metodi per lavorare sui dati presenti nel database.

Dipendenze

- IChatDataSource.

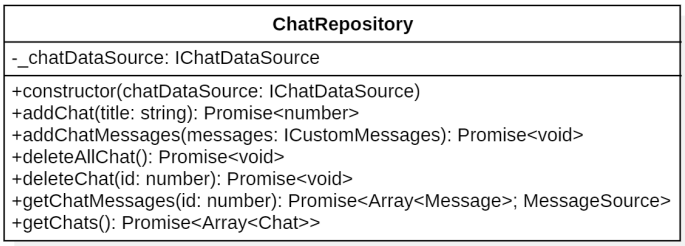

Figura 31: UML della classe ChatRepository

4.5.5 Data Sources

MinioDataSource

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_db: S3*.

Metodi

- *addOne(doc: Document, model: IModel): Promise<void>*
  Metodo per l'aggiunta di un nuovo documento nella collezione di un modello;
- *deleteOne(docName: string, model: IModel): Promise<void>*
  Metodo per l'eliminazione di un documento dalla collezione di un modello;
- *updateOne(docName:string, model:IModel, tag:Metadatas): Promise<void>*
  Metodo per l'aggiornamento del tag di visibilità di un documento di una particolare collezione;
- *getAll(model: IModel): Promise<Document[]>*
  Metodo per recuperare i dati dei documenti contenuti in una collezione;
- *getContent(docName: string, model: IModel): Promise<string>*
  Metodo per recuperare il link di visualizzazione di un documento.

Descrizione

La classe MinioDataSource implementa concretamente i metodi esposti all'interno di DocumentRepository. È lei ad operare sul database, tramite un attributo di tipo *S3*.

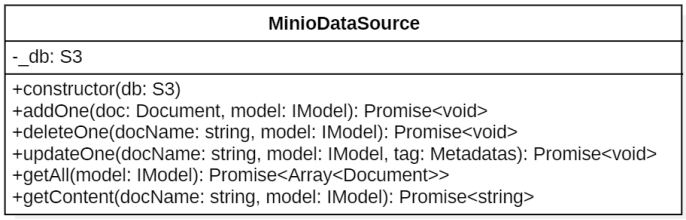

Figura 32: UML della classe MinioDataSource

ChromaDataSource

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_vDb: ChromaClient*.

Metodi

- *addOne(embeddings: Embeddings, model: IModel): Promise<void>*
  Metodo per aggiungere gli embedding di un nuovo documento alla collezione di vettori di un modello;
- *deleteOne(ids: string[], model: IModel): Promise<void>*
  Metodo per eliminare gli embedding di un preciso documento dalla collezione di vettori di un modello;
- *updateOne(metadatas: Metadatas, model: IModel, ids: string[]): Promise<void>*
  Metodo per modificare i metadati degli embedding di un preciso documento in una collezione di vettori di un modello;
- *getIds(docName: string, model: IModel): Promise<string>*
  Metodo per recuperare gli id di tutti i vettori che si riferiscono ad un preciso documento in una collezione di vettori di un modello.

Descrizione

La classe ChromaDataSource implementa concretamente i metodi esposti all'interno di EmbeddingRepository. È lei a svolgere le operazioni sul database vettoriale, tramite un attributo di tipo *ChromaClient*.

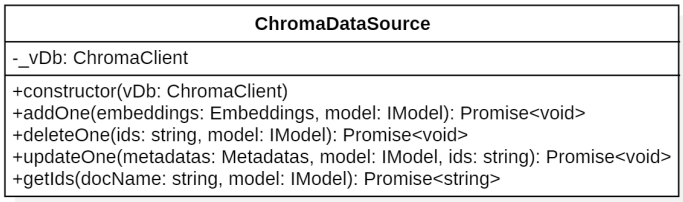

Figura 33: UML della classe ChromaDataSource

PostgresDataSource

Decoratori

- *@injectable()*: decoratore per la Dependency Injection mediante Tsyringe.

Attributi

- private readonly *_dB: Pool*.

Metodi

- *addMessages(messages: ICustomMessages): Promise<void>*
  Metodo per salvare un messaggio nella relativa chat history di una conversazione;
- *addOne(title: string): Promise<number>*
  Metodo per creare una nuova sessione di conversazione;
- *deleteAll(): Promise<void>*
  Metodo per eliminare tutte le sessioni di conversazione;
- *deleteOne(id: number): Promise<void>*
  Metodo per eliminare la chat history di una sessione;
- *getAll(): Promise<Chat[]>*
  Metodo per recuperare le sessioni attive;
- *getAllMessages(id: number): Promise<allMessages: Message[], source: MessageSource>*
  Metodo per recuperare la chat history di una sessione.

Descrizione

La classe PostgresDataSource implementa concretamente i metodi esposti all'interno di ChatRepository. È lei a svolgere le operazioni sul database Postgres, tramite un attributo di tipo *Pool*.

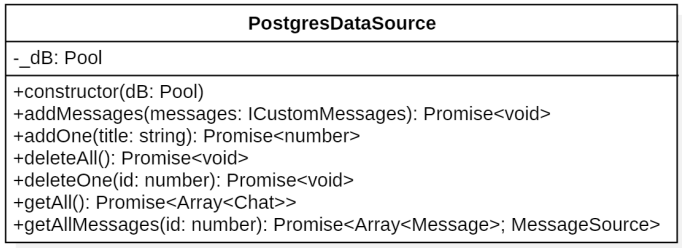

Figura 34: UML della classe PostgresDataSource

4.5.6 Basi di dati

ChromaDB

Descrizione
ChromaDB rappresenta il database vettoriale adibito a contenere gli embedding dei documenti caricati nel sistema. Esso contiene due collection di vettori, ovvero una per ogni modello utilizzato nella produzione e consultazione degli embedding.
All'interno di ogni collection sono contenuti le informazioni riguardanti gli embedding dei documenti.
In particolare, ad ogni oggetto nella collection è associato un valore identificativo ids di tipo *string*, il nome del documento, i metadati ad esso associato e l'effettivo vettore di numeri reali ottenuto dall'embedding del documento.

Requisiti associati

- RFO-27;
- RFO-29;
- RFD-36;
- RFD-37;
- RFO-55;
- RFO-56;
- RFD-57;
- RFO-64;
- RFO-65;
- RIO-4.

MinIO

Descrizione
MinIO rappresenta il database adibito a contenere i documenti caricati nel sistema. Esso contiene due bucket di oggetti rappresentanti i documenti, ovvero uno per ogni modello supportato nell'utilizzo dell'applicazione.
All'interno di ogni bucket sono contenuti i documenti, i quali sono rappresentati come oggetti.
In particolare, ad ogni oggetto documento nel bucket è associato un nome, una dimensione, la data di inserimento e dei tag. I singoli documenti sono consultabili grazie alla generazione di un link di visualizzazione.

Requisiti associati

- RFO-3;

- RFO-4;
- RFO-5;
- RFO-6;
- RFZ-7;
- RFO-8;
- RFD-9;
- RFO-10;
- RFO-27;
- RFO-29;
- RIO-2;
- RIO-3.

Postgres

Descrizione

Postgres è il database adibito a contenere le chat history di ogni sessione di conversazione attiva nel sistema.
Sono definite due diverse tabelle: la tabella *chat_threads*, che rappresenta le sessioni di conversazione, e la tabella *messages*, che rappresenta e contiene tutte le informazioni dei singoli messaggi. *Chat_threads* è composto da un valore identificativo numerico seriale *id*, da una stringa di testo *title* (il nome della sessione) e da un TIMESTAMP *created_at*.
Messages è invece identificata da un *id* VARCHAR e contiene un *thread_id* che si riferisce all'id di un *chat_threads*, dal contenuto testuale *content*, dal *role* (chi ha scritto il messaggio), *created_at*, *sourcePage* (numero di pagina del documento fonte, se il messaggio è una risposta) e *sourceLink* (link del documento fonte, se il messaggio è una risposta).

Requisiti associati

- RFD-48;
- RFD-49;
- RFD-50;
- RFO-52;
- RFO-53;
- RFO-59;
- RFD-60;
- RFO-63.

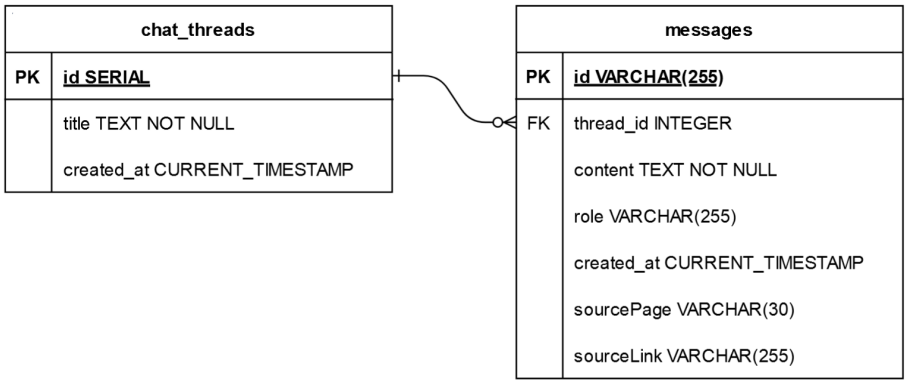

Figura 35: Diagramma ER del database Postgres

```
CREATE TABLE IF NOT EXISTS chat_threads (
        id SERIAL PRIMARY KEY,
        title TEXT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS messages (
        id VARCHAR(255) PRIMARY KEY,
        thread_id INTEGER REFERENCES chat_threads(id) ON DELETE CASCADE,
        content TEXT NOT NULL,
        role VARCHAR(255),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        sourcePage VARCHAR(30),
        sourceLink VARCHAR(255),
        UNIQUE (thread_id, id)
);
```

Figura 36: Query per la creazione del database Postgres

4.5.7 API

Descrizione
Per la funzionalità di chat, è stata implementata un'api apposita. Il body della richiesta contiene l'intera chat history della sessione nella quale si sta svolgendo la conversazione, oltre al modello da utilizzare nelle operazioni di retrieve delle informazioni. Ottenuta la domanda, prendendo l'ultimo messaggio della conversazione, e il contesto della chat history con tutti gli altri, viene completato il prompt da porre al modello. Terminata la fase di retrive, se individuata la risposta essa sarà fornita come streaming di testo accompagnata dai riferimenti sul documento fonte, utilizzando i metadati ad esso associati e presenti nel database vettoriale.

Elementi
Ottenuti dal body della richiesta modello e chat history, viene definita la collezione e quale funzione di embedding utilizzare in base al modello selezionato, utilizzando il metodo *fromExistingCollection* di Chroma. Successivamente viene composto il prompt finale da utilizzare nella *RunnableSequence* di LangChain, definendo domanda, metodo di retrieve, la chat history e il modello da utilizzare nell'operazione.
Al termine del processo, ottenuta risposta e vettore di embedding di riferimento, viene composta l'informazione sul documento fonte della risposta e in seguito restituite come risposta con uno streaming di testo.
In caso di esito negativo della chiamata, viene ritornata una risposta con status 500, contenente nel body il messaggio di errore sollevato.

Endpoint: api/chat
Metodo HTTP: POST
Esito positivo
Status: 200;
Body: Streaming di testo della risposta, come da descrizione.
Esito negativo
Status: 500;
Body: error: messaggio di errore sollevato.

Requisiti associati

- RFO-44;
- RFO-45;
- RFO-46;
- RFO-64;
- RFO-65.

4.6 Design Pattern utilizzati

4.6.1 Dependency Injection

Descrizione

Al fine di minimizzare le dipendenze delle classi implementate per il back-end del prodotto, è stato adottato il Dependency Injection pattern.
Lo scopo di questo pattern è separare il comportamento di una componente dalla risoluzione delle sue dipendenze, così da ridurne il grado di accoppiamento.

Implementazione

Nell'implementazione di questo pattern, è stata utilizzata Tsyringe, una libreria di Dependency Injection per Typescript. Attraverso l'utilizzo dei decoratori @injectable e @inject, è possibile dichiarare le classi le cui dipendenze sono iniettabili dal container e le singole dipendenze.

```
@injectable()
class AddDocumentController {
  private readonly _useCase: AddDocumentUsecase;

  constructor(@inject("addDocUsecase") useCase: AddDocumentUsecase) {
    this._useCase = useCase;
  }

...

}
```

Figura 37: Esempio di implementazione nel prodotto di @injectable e @inject

Nell'esempio sopra riportato, "addDocUsecase" è il token associato alla classe dichiarata come dipendenza nel costruttore, e sarà utilizzato per registrare tale dipendenza nel container che si occuperà di iniettarla.
Una volta registrato, quando si chiederà di risolvere le dipendenze del controller, esse saranno risolte tramite Dependency Injection.

```
container.register<AddDocumentUsecase>("addDocUsecase", {
  useClass: AddDocumentUsecase,
});

...

const addDocumentController = container.resolve(AddDocumentController);
export addDocumentController;
```

Figura 38: Esempio di registrazione e risoluzione delle dipendenze nel container Tsyringe

4.6.2 Controller-Service-Repository

Descrizione

Coerentemente ai principi dell'architettura clean adottata per il sistema, per il back-end è stato utilizzato il CSR pattern. La business logic è implementata all'interno dei Service, che utilizzano i Repository per operare sui dati in lettura e scrittura. Adibiti ad elaborare le richieste e chiamare i Service, i Controller restituiscono i risultati delle operazioni al front-end.

Implementazione

Nell'implementazione di questo pattern, sono state definite delle classi Controller, Usecase e Repository.
Il Controller, come facilmente intuibile, riceve le richieste provenienti dal front-end, chiama lo Usecase associato e gestisce eventuali errori nell'esecuzione delle operazioni di business. Le classi Usecase rappresentano i Service, ovvero contengono la logica di business dell'applicazione, necessaria per l'esecuzione delle varie funzionalità. Ad essere chiamate dal Service sono le Repository, classi responsabili della gestione della persistenza dei dati. Nascondono i dettagli della persistenza ai livelli superiori dell'applicazione, offrendo delle interfacce per recuperare e salvare i dati.
L'effettiva logica di accesso ai dati è delegata ai DataSource.

```
@injectable()
class DeleteDocumentController {
  private readonly _useCase: DeleteDocumentUsecase;

  constructor(@inject("delDocUsecase") useCase: DeleteDocumentUsecase) {
    this._useCase = useCase;
  }
  async handle(docName: string, model: IModel): Promise<Response> {
    try {
      await this._useCase.execute({ docName: docName, model: model });
    ...
  }
...
}

@injectable()
class DeleteDocumentUsecase
  implements IUsecase<{ docName: string; model: IModel }, void>
{
  private readonly _documentRepository: IDocumentRepository;
  private readonly _embeddingRepository: IEmbeddingRepository;

  constructor(
    @inject("documentRepository") documentRepository: IDocumentRepository,
    @inject("embeddingRepository") embeddingRepository: IEmbeddingRepository,
  ) {
    this._documentRepository = documentRepository;
    this._embeddingRepository = embeddingRepository;
  }
  async execute({ docName, model }: { docName: string; model: IModel }) {
    await this._documentRepository.deleteDocument(docName, model);
    ...
    await this._embeddingRepository.deleteEmbedding(ids, model);
  }
}

@injectable()
class DocumentRepository implements IDocumentRepository {
  private _documentDataSource: IDocumentDataSource;

  constructor(
    @inject("documentDataSource") documentDataSource: IDocumentDataSource,
  ) {
    this._documentDataSource = documentDataSource;
  }
  ...
  async deleteDocument(docName: string, model: IModel): Promise<void> {
    await this._documentDataSource.deleteOne({docName: docName,model: model});
  }
  ...
}
```

Figura 39: Esempio di implementazione del CSR pattern per l'operazione di eliminazione di un documento

4.6.3 Compound Components

Descrizione

Specifico di React e di applicazioni components-based, il Compound Components è un pattern che prevede l'elevata modularizzazione del codice front-end, andando a definire singoli componenti, con un unico scopo, da inserire all'interno di componenti contenitori. Con questo approccio, viene creata un'alberatura delle componenti che prevede una relazione di padre-figlio fra le stesse.
Seguendo questo pattern, è possibile creare componenti riutilizzabili, che aumentano la flessibilità e la manutenibilità codice, evitando di codificare l'intera interfaccia grafica in un'unica grande componente.

Implementazione

Per attenersi al pattern, è stato necessario suddividere le singole pagine dell'applicazione in molte componenti. Per aiutarci in questo compito, è stato utilizzato il framework di componenti Shadcn/ui, implementando numerose componenti personalizzabili e altamente riutilizzabili, componendo modularmente la GUI finale.

4.6.4 Container-Presentational Components

Descrizione

Anch'esso tipico di React e di web app basate su componenti, il Container Presentational è un design pattern strettamente legato al Compounds Component. Esso prevede la separazione dei componenti con solo logica di presentazione da quelli che devono anche gestire la logica e lo stato dei componenti.
L'utilizzo di questo design pattern, ad affiancare il già esposto Compound Components, prevede che i componenti così detti foglia, ovvero che non sono padre di altri componenti figli, non presentino codice per la gestione del loro stato. Esso dev'essere infatti presente nel "container component", ovvero il componente padre che lo contiene.
Utilizzando questo pattern si favorisce la modularità del codice, così come la separazione e distinzione del codice in base alla logica e responsabilità a cui devono rispondere.

Implementazione

Anche per questo pattern l'utilizzo di Shadcn/ui si è rivelato prezioso, in quanto è stato possibile implementare facilmente componenti prefabbricate, personalizzabili e

riutilizzabili con puro scopo di presentazione, dovendo solo implementare la gestione dello stato dei componenti in quelli ”container” o radice.