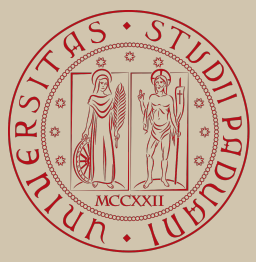
Specifica Tecnica

2025-03-27

V0.0.3

sweetenteam@gmail.com

https://sweetenteam.github.io



Destinatari	Prof. Tullio Vardanega Prof. Riccardo Cardin AzzurroDigitale
Redattori	Orlando Ferazzani
Verificatori	Mouad Mahdi blank

Registro delle modifiche

Versione	Data	Autori	Verificatori	Dettaglio
0.0.3	2025-03-27	Orlando Ferazzani	blank	Aggiunte tecnologie di testing e miglioramenti generali
0.0.2	2025-03-15	Orlando Ferazzani	blank	Aggiunta sezione «Architettura frontend»
0.0.1	2025-02-27	Orlando Ferazzani	Mouad Mahdi	Prima stesura documento

Indice

1) **Introduzione** 5
1.1) Scopo del documento 5
1.2) Scopo del prodotto 5
1.3) Miglioramenti e maturità 5
1.4) Glossario 5
1.5) Riferimenti 6
1.5.1) Riferimenti normativi 6
1.5.2) Riferimenti informativi 6
1.5.3) Riferimenti Tecnici 6
2) Tecnologie 6
2.1) Tecnologie di sviluppo 7
2.1.1) Typescript 7
2.1.2) Langchain 7
2.1.3) Node.js 7
2.1.4) Nest.js 7
2.1.5) GroqCloud 7
2.1.6) Qdrant 8
2.1.7) NomicAi 8
2.1.8) PostgreSQL 8
2.1.9) Octokit 8
2.1.10) JiraJs 8
2.1.11) ConfluenceJs 8
2.1.12) Docker 9
2.1.13) React.js 9
2.1.14) ReactQuery 9
2.1.15) TailwindCSS 9
2.1.16) Next.js 9
2.1.17) ShadCn 9
2.1.18) LucideReact 10
2.2) Tecnologie di testing 10
2.2.1) Jest 10
2.2.2) ESLint 10
3) Architectura di sistema 10
3.1) Architettura frontend 10

Lista della immagini

Figura 1 Logo BuddyBot 5
Figura 2 Logo Typescript 7
Figura 3 Logo di Langchain 7
Figura 4 Logo di Node.js 7
Figura 5 Logo di Nest.js 7
Figura 6 Logo di GroqCloud 7
Figura 7 Logo di Qdrant 8
Figura 8 Logo di NomicAi 8
Figura 9 Logo di PostgreSQL 8
Figura 10 Logo di Octokit 8
Figura 11 Logo di JiraJs 8
Figura 12 Logo di ConfluenceJs 8
Figura 13 Logo di Docker 9
Figura 14 Logo di ReactJs 9
Figura 15 Logo di ReactQuery 9
Figura 16 Logo di TailwindCSS 9
Figura 17 Logo di Next.js 9
Figura 18 Logo di ShadCn 10
Figura 19 Logo di LucideReact 10
Figura 20 Logo di Jest 10
Figura 21 Logo di ESLint 10
Figura 22 Header della pagina in dark mode 11
Figura 23 Navbar della pagina in dark mode 11
Figura 24 ChatWindow della pagina in dark mode 11
Figura 25 InputForm della pagina in dark mode 12
Figura 26 Bubble della pagina in dark mode 12
Figura 27 Diagramma UML dell'architettura di frontend 19

1) Introduzione

1.1) Scopo del documento

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

Figura 1: Logo BuddyBot

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

Il documento ha quindi scopi molteplici:

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

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

1.2) Scopo del prodotto

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

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

1.3) Miglioramenti e maturità

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

1.4) Glossario

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

1.5) Riferimenti

1.5.1) Riferimenti normativi

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

1.5.2) Riferimenti informativi

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

1.5.3) Riferimenti Tecnici

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

2) Tecnologie

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

2.1) Tecnologie di sviluppo

2.1.1) Typescript

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

Figura 2: Logo Typescript

2.1.2) Langchain

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

Figura 3: Logo di Langchain

2.1.3) Node.js

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

Figura 4: Logo di Node.js

2.1.4) Nest.js

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

Figura 5: Logo di Nest.js

2.1.5) GroqCloud

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

Figura 6: Logo di GroqCloud

2.1.6) Qdrant

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

Figura 7: Logo di Qdrant

2.1.7) NomicAi

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

Figura 8: Logo di NomicAi

2.1.8) PostgreSQL

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

Figura 9: Logo di PostgreSQL

2.1.9) Octokit

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

Figura 10: Logo di Octokit

2.1.10) JiraJs

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

Figura 11: Logo di JiraJs

2.1.11) ConfluenceJs

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

Figura 12: Logo di ConfluenceJs

2.1.12) Docker

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

Figura 13: Logo di Docker

2.1.13) React.js

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

Figura 14: Logo di ReactJs

2.1.14) ReactQuery

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

Figura 15: Logo di ReactQuery

2.1.15) TailwindCSS

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

Figura 16: Logo di TailwindCSS

2.1.16) Next.js

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

Figura 17: Logo di Next.js

2.1.17) ShadCn

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

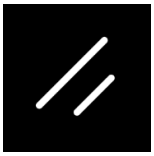


Figura 18: Logo di ShadCn

2.1.18) LucideReact

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

Figura 19: Logo di LucideReact

2.2) Tecnologie di testing

2.2.1) Jest

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

Figura 20: Logo di Jest

2.2.2) ESLint

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

Figura 21: Logo di ESLint

3) Architectura di sistema

3.1) Architettura frontend

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

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

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

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

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

- `Header.tsx`;

Figura 22: Header della pagina in dark mode

- `Navbar.tsx`;

Figura 23: Navbar della pagina in dark mode

- `ChatWindow.tsx`;

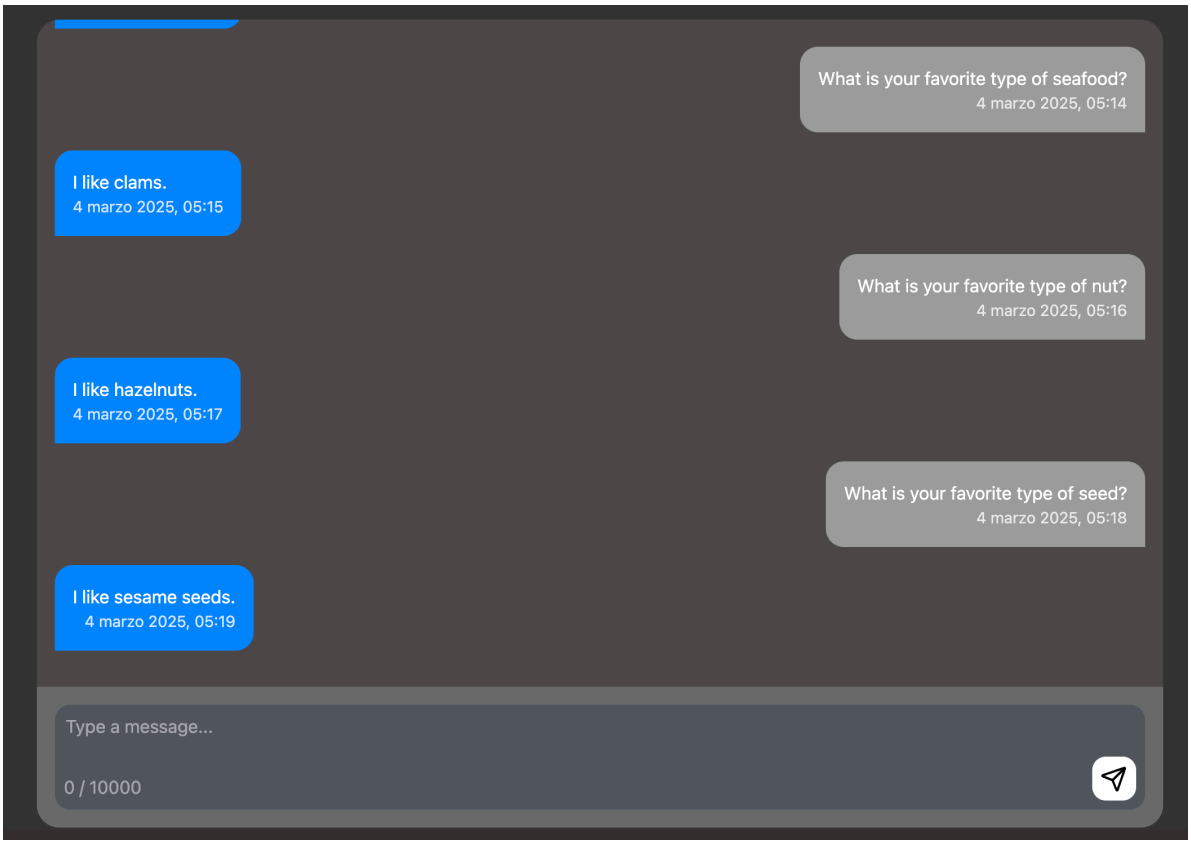


Figura 24: ChatWindow della pagina in dark mode

- `InputForm.tsx`;

Figura 25: InputForm della pagina in dark mode

- `Bubble.tsx`;

Figura 26: Bubble della pagina in dark mode

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

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

Architettura in dettaglio

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

- ```
  //Action.ts -> action che l'app può fare
    import { Message } from "@/types/Message";
  import { QuestionAnswer } from "./QuestionAnswer";

  export type ChatAction =
    | { type: "LOAD_HISTORY_START" }
    | { type: "LOAD_HISTORY_SUCCESS"; payload: QuestionAnswer[], hasMore: boolean }
    | { type: "LOAD_HISTORY_ERROR" }
    | { type: "ADD_MESSAGE_START"; id: string, question: Message }
    | { type: "ADD_MESSAGE_SUCCESS"; id: string, answer: Message, newid: string }
    | { type: "ADD_MESSAGE_ERROR"; id: string }
    | { type: "SCROLL_DOWN" };
  ```
- ```
  //ChatContext.ts -> contesto della chat
  import { createContext } from "react";
  import { ChatAction } from "./Action";
  import { ChatState } from "./ChatState";

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

  export const ChatContext = createContext<ChatContext | undefined>(undefined);
  ```
- ```
  //ChatProviderProps.ts -> props del provider della chat
    import { Target } from "@/adapters/Target";
  import { ReactNode } from "react";

  export interface ChatProviderProps {
      children: ReactNode;
      adapter: Target;
  }
  ```
- ```
  //ChatState.ts -> stato della chat
  import { QuestionAnswer } from "./QuestionAnswer";

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

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

- ```
  //Message.ts -> messaggio
  export interface Message {
      content: string;
      timestamp: number;
  }
  ```
- ```
  //QuestionAnswer.ts -> domanda e risposta
  import { Message } from "./Message";

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

Il frontend di BuddyBot inoltre utilizza dei `Reducers`, `Adapters` e `Providers` di modo che tutte le logiche siano disponibili e separate. Di seguito una breve spiegazione di come sono implementati nei componenti principali:

- ```
  //ThemeProvider.tsx = provider per il tema della pagina

  "use client";

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

  export function ThemeProvider({ children, ...props }: ThemeProviderProps) {
      return (
          <NextThemesProvider {...props}>
              {children}
          </NextThemesProvider>
      );
  }
  ```
- ```
  //ChatProvider.tsx = provider per la chat

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

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

      const loadHistory = async (): Promise<void> => {
        dispatch({ type: "LOAD_HISTORY_START" });
        try {
          if (state.messages.length === 0) {
            const olderMessages: QuestionAnswer[] = await adapter.requestHistory("",
  10);
            dispatch({ type: "LOAD_HISTORY_SUCCESS", payload: olderMessages,
  ```

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

    const sendMessage = async (text: string) => {
      const id = generateId();
      const newMessage: Message = {
        content: text,
        timestamp: Date.now(),
      };
      dispatch({ type: "ADD_MESSAGE_START", id: id, question: newMessage });
      dispatch({ type: "SCROLL_DOWN" });
      try {
        const botResponse: { answer: Message, id: string } = await
adapter.requestAnswer(newMessage);
        dispatch({ type: "ADD_MESSAGE_SUCCESS", id: id, answer: botResponse.answer,
newid: botResponse.id });
      }
      catch (error) {
        dispatch({ type: "ADD_MESSAGE_ERROR", id: id });
      }
    };

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

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

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

Questi provider vengono utilizzati, rispettivamente, all'interno dei file `layout.tsx` e `chatWindow.tsx` per gestire il tema della pagina e la chat stessa.

Il Reducer e gli Adapters invece:

- ```
  //ChatReducer.ts
  import { Message } from "@/types/Message";
  import { ChatState } from "@/types/ChatState";
  import { ChatAction } from "@/types/Action";

  export const chatReducer = (state: ChatState, action: ChatAction): ChatState => {
    switch (action.type) {
      case "LOAD_HISTORY_START":
        return {
          ...state,
          loadingHistory: true,
          errorHistory: false,
          hasMore: false,
        };
      case "LOAD_HISTORY_SUCCESS":
        return {
          ...state,
          messages: [...action.payload, ...state.messages],
          loadingHistory: false,
          errorHistory: false,
          hasMore: action.hasMore,
        };
      case "LOAD_HISTORY_ERROR":
        return {
          ...state,
          loadingHistory: false,
          errorHistory: true,
          hasMore: false,
        };
      case "ADD_MESSAGE_START":
        return {
          ...state,
          messages: [...state.messages, { id: action.id, question: action.question,
  answer: {} as Message, error: false, loading: true }],
        };
      case "ADD_MESSAGE_SUCCESS":
        const updatedMessagesSuccess = state.messages.map((msg) => {
          if (msg.id === action.id) {
            return {
              ...msg,
              id: action.newid,
              answer: action.answer,
              loading: false,
              error: false,
            };
          }
          return msg;
        });
        return {
          ...state,
          messages: updatedMessagesSuccess,
        };
      case "ADD_MESSAGE_ERROR":
        const updatedMessagesError = state.messages.map((msg) => {
          if (msg.id === action.id) {
  ```
  ```

```
            return {
              ...msg,
              loading: false,
              error: true,
            };
          }
          return msg;
        });
        return {
          ...state,
          messages: updatedMessagesError,
        };
      case "SCROLL_DOWN":
      return {
        ...state,
        hasToScroll: !state.hasToScroll,
      };
      default:
        return state;
    }
  };
```

- ```
  //adapter.ts
    import { QuestionAnswer } from "@/types/QuestionAnswer";
  import { Message } from "@/types/Message";
  import { Target } from "./Target";
  import { AdapterFacade } from "./AdapterFacade";
  import { generateId } from "@/utils/generateId";

  export class Adapter implements Target {
      private adapterFacade: AdapterFacade;

      constructor() {
          this.adapterFacade = new AdapterFacade();
      }

      async requestHistory(id: string, offset: number): Promise<QuestionAnswer[]> {
          try {
              const jsonResponse = await this.adapterFacade.fetchHistory(id, offset);
              return this.adaptQuestionAnswerArray(jsonResponse);
          } catch (error) {
              throw new Error("Error fetching history");
          }
      }
      async requestAnswer(question: Message): Promise<{ answer: Message; id:
  string; }> {
          try {
              const answer = await
  this.adapterFacade.fetchQuestion(this.adaptMessageToJSON(question));
              return {
                  answer: this.adaptMessage(answer.answer),
                  id: answer.id
              };
          } catch (error) {
              throw new Error("Error fetching history");
          }
      }
  ```
  ```

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

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

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

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

- ```
  //AdapterFacade.ts
  ```

```
  import historyData from "@/json/history.json";
import historyData1 from "@/json/history1.json";
import { generateId } from "@/utils/generateId";

export class AdapterFacade {
    async fetchHistory(id: string, offset: number): Promise<any[]> {
        const controller = new AbortController();
        const timeoutId = setTimeout(() => controller.abort(), 3000);
        if (id === "") return historyData1;
        else if (id == "240") return historyData;
        return [];

        try {
            const response = await fetch(`https://api.example.com/history`, {
                method: "POST",
                headers: {
                    "Content-Type": "application/json",
                    // Eventuali header necessari
                },
                body: JSON.stringify({ id, offset }),
                signal: controller.signal,
            });
            clearTimeout(timeoutId);
            if (!response.ok) throw new Error("Error fetching history");
            return await response.json();
        } catch (error) {
```
```

```
            throw new Error("Error fetching history");
        }
    }

    async fetchQuestion(data: any): Promise<any> {
        const controller = new AbortController();
        const timeoutId = setTimeout(() => controller.abort(), 3000);
        return { answer: { content: data.question, timestamp: data.timestamp }, id:
generateId() };

        try {
            const response = await fetch(`https://api.example.com/send`, {
                method: "POST",
                headers: {
                    "Content-Type": "application/json",
                    // Eventuali header necessari
                },
                body: JSON.stringify(data),
                signal: controller.signal,
            });
            clearTimeout(timeoutId);
            if (!response.ok) throw new Error("Error sending message");
            return await response.json();
        } catch (error) {
            throw new Error("Error sending message");
        }
    }
}
```

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

  export interface Target {
    requestHistory(id: string, offset: number): Promise<QuestionAnswer[]>;
    requestAnswer(question: Message): Promise<{answer: Message, id: string}>;
  }
  ```

Questi file sono utilizzati per gestire la logica della chat e per adattare i dati ricevuti dalle API in modo che possano essere utilizzati all'interno della web app.
  ```

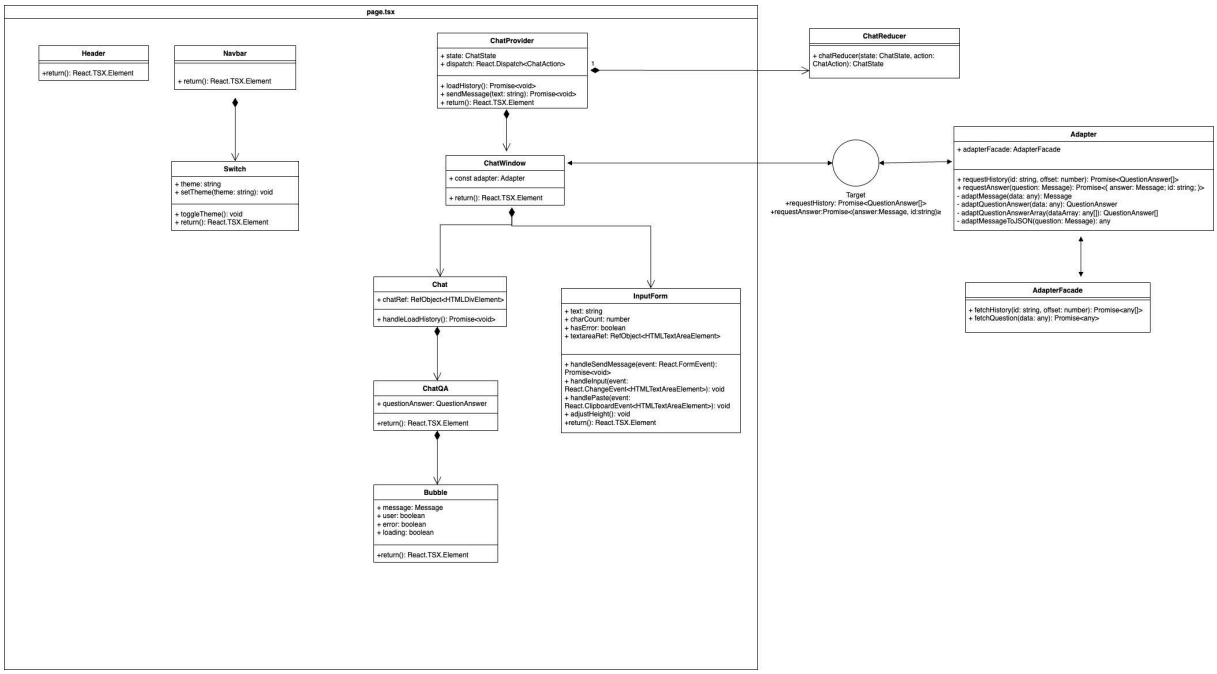


Figura 27: Diagramma UML dell'architettura di frontend