SkyNet

Specifica Architetturale

Bot4Me

---

skynet.swe@gmail.com

5 Settembre 2022

Redattori: Anna Cisotto Bertocco, Kaltrina Collaku

Verificatori: Alberto Matterazzo, Anna Cisotto Bertocco

Responsabile: Davide Sut

Destinatari: Prof. Tullio Vardanega, Prof. Riccardo Cardin, Imola Informatica

Uso: **Esterno**

Stato: In attesa di approvazione

Versione: **0.1.0**

Registro delle Modifiche

Versione	Autore (verificatore)	Data	Ruolo	Descrizione
0.1.0	Anna Cisotto Bertocco (-)	05-09-2022	Verificatore	Verifica documento
0.0.6	Davide Sut (Anna Cisotto Bertocco)	05-09-2022	Responsabile	Stesura §3.2
0.0.5	Kaltrina Collaku(Anna Cisotto Bertocco)	05-09-2022	Progettista	Stesura §1 e §3.1
0.0.4	Kaltrina Collaku(Alberto Matterazzo)	01-09-2022	Programmatore	Stesura §4
0.0.3	Anna Cisotto Bertocco (Alberto Matterazzo)	30-08-2022	Programmatore	Stesura §2.1, §2.2 e §3.2.3
0.0.2	Anna Cisotto Bertocco (Alberto Matterazzo)	29-08-2022	Programmatore	Stesura §3.3
0.0.1	Anna Cisotto Bertocco (Davide Dinato)	16-08-2022	Amministratore	Creazione struttura documento

Indice

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

Questo documento illustra pianificazione e modalità di sviluppo relativi alle specifiche architetturali adottate per la realizzazione del progetto. In particolare, il documento si articola in due macro-sezioni principali:

- **Architettura**: in questa sezione viene illustrata l'architettura del progetto, tramite diagrammi delle classi e di sequenza;
- **Tecnologie**: in questa sezione vengono esposte le tecnologie che il gruppo ha scelto di adottare per la realizzazione del progetto.

Vi è poi un'ulteriore sezione dedicata ad illustrare il *setup* iniziale dell'applicativo per permetterne il corretto utilizzo.

1.2 Scopo del progetto

Lo scopo del progetto è quello di semplificare le attività aziendali di routine mediante l'utilizzo di un $ChatBot_G$, rendendo possibile un'interazione sia testuale che vocale con i dipendenti di Imola Informatica. Il $ChatBot_G$ assisterà i dipententi nelle attività che richiedono loro di interfacciarsi con diversi servizi e applicativi; le operazioni principali sono:

- tracciamento della presenza in sede;
- inserimento di una nuova attività da consuntivate.

L'applicativo finale sarà una $Web\ App_G$ accessibile sia da dispositivi $mobile_G$, quali smartphone e tablet, sia da dispositivi $desktop_G$ tramite $browser_G$.

1.3 Glossario

Per evitare incomprensioni e ambiguità durante la lettura del documento, vengono utilizzati due simboli a pedice di alcuni termini, con le seguenti funzioni:

- G per indicare i termini la cui definizione si trova nel Glossario v3.0.0$_D$
- D per indicare il nome di un documento esterno

1.4 Riferimenti

1.4.1 Normativi

- Norme di Progetto v1.0.0$_D$
- Piano di progetto v1.1.0$_D$

1.4.2 Informativi

- Slide del corso - Diagrammi delle classi (UML)
  https://www.math.unipd.it/ rcardin/swea/2021/Diagrammi%20delle%20Classi_4x4.pdf
- Slide del corso - Diagrammi di sequenza:
  https://www.math.unipd.it/ rcardin/swea/2022/Diagrammi%20di%20Sequenza.pdf
- Slide del corso - Pattern architetturali:
  https://www.math.unipd.it/ rcardin/swea/2022/Software%20Architecture%20Patterns.pdf
- Chatterbot - documentazione
  https://chatterbot.readthedocs.io/en/stable/

- Django - documentazione
  https://docs.djangoproject.com/en/3.2/intro/
- React - documentazione
  https://it.reactjs.org/docs/getting-started.html

2 Tecnologie di sviluppo

In questa sezione vengono elencate e descritte le tecnologie, le librerie e i linguaggi di programmazione utilizzati per lo sviluppo delle componenti del *front-end*$_G$ e del *back-end*$_G$.

2.1 Front-end$_G$

2.1.1 React$_G$

React$_G$ è una libreria *open-source*$_G$ Javascript$_G$ utilizzata per creare interfacce utente.
È stata utilizzata per la creazione dell'UI della *Web App*$_G$.

- **Versione usata:** 1.0.0
- **Documentazione:** https://it.reactjs.org/docs/getting-started.html

2.1.2 HTML5$_G$

HTML5$_G$ è uno linguaggio di *markup* e standard *W3C*$_G$ utilizzato per la struttura di pagina web.
È stato utilizzato per la struttura della pagina web attraverso cui l'utente può interfacciarsi con il *ChatBot*$_G$.

- **Documentazione:** https://dev.w3.org/html5/spec-LC/

2.1.3 CSS3$_G$

CSS3$_G$ è un linguaggio di formattazione per documenti *HTML5*$_G$.
È stato utilizzato per la creazione dello stile grafico della pagina web attraverso cui l'utente può interfacciarsi con il *ChatBot*$_G$, utilizzando un approccio *mobile-first*.

- **Documentazione:** https://www.w3.org/TR/2001/WD-css3-roadmap-20010523/

2.1.4 Javascript$_G$

Javascript$_G$ è un linguaggio di programmazione orientato agli eventi, utilizzato per creare effetti dinamici interattivi nelle pagine web.
È stato utilizzato per gestire gli eventi lato *client* della *Web App*$_G$.

- **Documentazione:** https://www.w3schools.com/js/default.asp

2.2 Back-end$_G$

2.2.1 Python$_G$

Python$_G$ è un linguaggio di programmazione orientato agli oggetti, usato per la creazione di applicazioni distribuite.
È stato utilizzato per sviluppare il *back-end*$_G$ dell'applicativo.

- **Versione usata:** 3.7.9
- **Documentazione:** https://www.python.org/doc/

2.2.2 Django$_G$

Django$_G$ è un *web framework*$_G$ *open-source*$_G$ scritto in *Python*$_G$ per lo sviluppo di applicazioni web.
È stato utilizzato per esporre le *REST API*$_G$ con cui si interfaccia il *client* e per gestire le chiamate *HTTP*$_G$ ai servizi *REST*$_G$ del proponente Imola Informatica.

- **Versione usata:** 3.2.13
- **Documentazione:** https://docs.djangoproject.com/en/4.1/

2.2.3 Chatterbot$_G$

Chatterbot$_G$ è una libreria *Python*$_G$ usata per generare risposte automatiche agli input di un utente tramite algoritmi di *IA*$_G$.
È stata utilizzata per generare le risposte ed elaborare i messaggi del *ChatBot*$_G$, anche attraverso l'estensione dei *LogicAdapter* già presenti nella libreria al fine di gestire le risposte alle varie richieste effettuabili nel *ChatBot*$_G$.

- **Versione usata:** 1.0.4
- **Documentazione:** https://chatterbot.readthedocs.io/en/stable/

2.2.4 Render$_G$

Render$_G$ è un *Platform as a Service* su *cloud*$_G$ che supporta vari linguaggi di programmazione.
È stato utilizzato per il *deploy* della *Web App*$_G$.

- **Documentazione:** https://render.com/docs

3 Architettura di sistema

3.1 Front-end$_G$

3.1.1 Descrizione

Questa componente permette l'interazione dell'utente con il *ChatBot*$_G$. E' stata sviluppata usando il *framework*$_G$ *React*$_G$, e permette l'interazione sia da dispositivi mobili che da dispositivi *desktop*$_G$, utilizzando un design responsive e *mobile-first*.
L'interfaccia utente permette di fare le seguenti azioni:

- Iniziare una conversazione con il *ChatBot*$_G$;
- Richiede al *ChatBot*$_G$ di effettuare una specifica operazione, ricevendo le risposte dal *bot*;
- Effettuare il *login*$_G$ e il *logout*$_G$ dall'applicativo;

L'utente può effettuare tutte le operazioni senza dover conoscere il funzionamento dei sistemi EMT$_G$ di Imola Informatica, riuscendo quindi a svolgere le operazioni solamente comunicando in maniera semplice e discorsiva con il *ChatBot*$_G$.

3.1.2 Dipendenze esterne

- **React e React-dom:** libreria *framework*$_G$ front-end$_G$ per poter utilizzare *React*$_G$;
- **@types/node, @types/react, @types/react-dom parcel:** permette di utilizzare typescript per aggiungere tipizzazione al codice;
- **Tailwind CSS, PostCSS, Autoprefix:** per permettere l'uso di stili CSS forniti da Tailwind CSS.

3.1.3 Diagramma delle classi

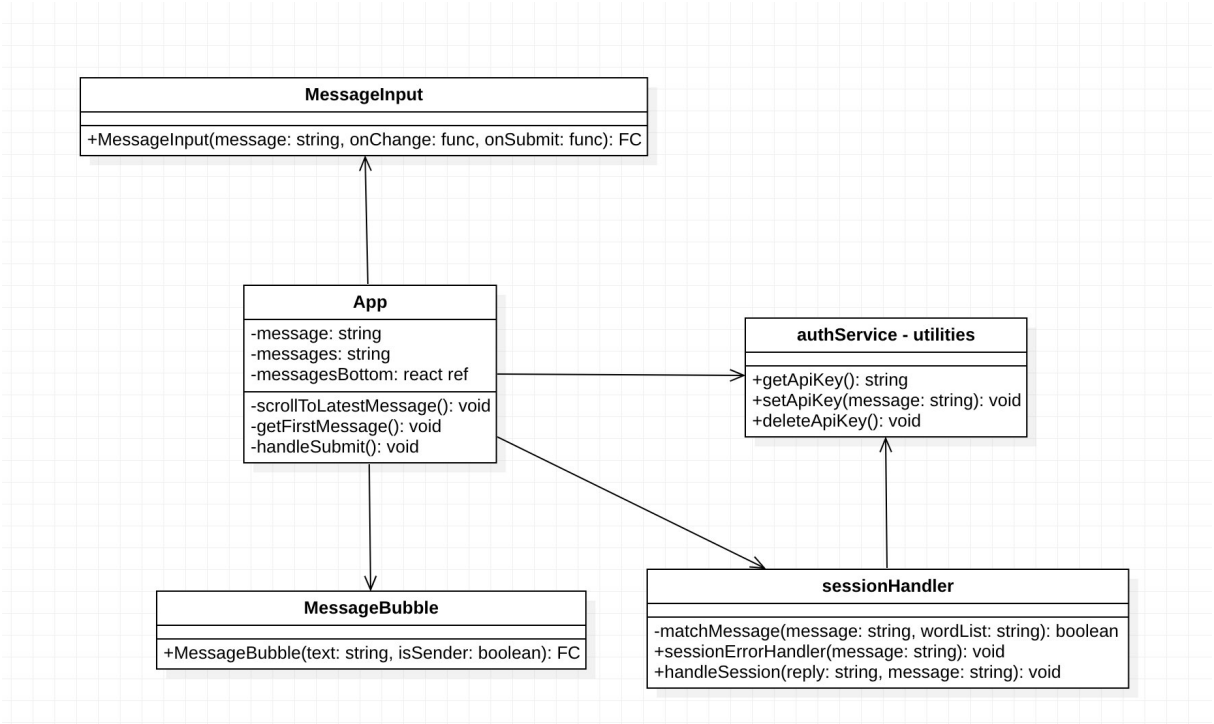

Figura 1: Diagramma delle classi per la parte *front-end*$_G$ dell'applicazione

3.1.3.1 App

Componente principale composto da:

- uno stato di messaggi inviati e ricevuti;
- il messaggio attuale che l'utente sta scrivendo;
- un riferimento alla posizione dell'ultimo messaggio nella lista dei messaggi (utile per spostare la conversazione all'ultimo messaggio ricevuto come avviene in altre app di messaggistica).

Questo componente utilizza ulteriori componenti:

- **MessageInput:** per mostrare all'utente l'input testuale;
- **MessageBubble:** per mostrare ogni messaggio nella conversazione tenuto in memoria.

Contiene questi metodi:

- **ScrollToLatestMessage:** permette di spostare lo scroll dell'utente all'ultimo messaggio;
- **GetFirstMessage:** permette di richiedere al back-end$_G$ il primo messaggio da mostrare all'utente per iniziare la confersazione;
- **HandleSubmit:** permette di inviare il messaggio scritto dall'utente al front-end$_G$.

Inoltre vengono utilizzate le componenti *authService* e *sessionHandler* durante l'invio dei messaggi per tenere traccia della sessione utente.

3.1.3.2 MessageBubble

Componente funzionale di *React*$_G$ che mostra un messaggio di uno degli interlocutori della *chat*$_G$. Ha come parametri:

- **text:** il messaggio da mostrare;
- **isSender:** variabile booleana che permette di mostrare il messaggio con uno stile grafico e posizione differente a seconda che sia un messaggio inviato dall'utente o inviato dal *ChatBot*$_G$.

3.1.3.3 MessageInput

Componente funzionale di *React*$_G$ che mostra un input testuale all'utente e riceve come parametri il messaggio scritto dall'utente e due funzioni:

- **onChange:** permette di tenere aggiornato il valore del messaggio;
- **onSubmit:** permette di inviare un segnale di *callback* quando l'utente invia il messaggio.

3.1.4 Sequenza di azioni

- Accesso alla chat
  1. Si accede all'indirizzo web della pagina.
  2. Non appena la pagina è stata caricata, avviene il caricamento del componente react *App* che effetua una chiamata HTTP$_G$ GET (fetch call) all'*Endpoint*$_G$ "/api/chatterbot" con il metodo *getFirstMessage()*, per iniziare la sessione e per mostrare un messaggio di benvenuto all'utente.
- Conversazione con il *ChatBot*$_G$
  1. L'utente scrive il messaggio nella casella di inserimento e preme il tasto per l'invio del messaggio.
  2. La richiesta di invio del messaggio viene gestita dalla funzione *handleSubmit()* in *App* che attraverso una fetch call, effettua una chiamata HTTP$_G$ POST all'*Endpoint*$_G$ "/api/chatterbot", per ottenere la risposta dal *ChatBot*$_G$.

3. Quando viene ritornato il risultato della chiamat, la risposta e il messaggio associato vengono mostrati nella chat.

- Inserimento della API Key$_G$

1. L'utente si deve ancora autenticare nell'applicativo e richiede di effettuare il *login*$_G$
2. Dopo aver ricevuto la risposta dal *ChatBot*$_G$, l'utente può inserire l'API Key$_G$ nella casella di invio dei messaggi
3. A questo punto viene effettuato l'invio della API key$_G$ al back-end$_G$ come un normale messaggio della conversazione
4. Se il back-end$_G$ riscontra la validità dell'API Key$_G$ inserita dall'utente invia un messaggio di conferma positiva del *login*$_G$.
5. Quando il front-end$_G$ riceve la conferma di *login*$_G$, viene salvata l'API Key$_G$ nel *Localstorage* (memoria persistente a lato *client*).
6. Nelle chiamate di rete successive al back-end verrà quindi sempre inserita nel *body*, assieme all'attributo "text", l'API Key$_G$ letta dal *Localstorage*.

3.2 Back-end$_G$

3.2.1 Descrizione

Questa componente è il *core* del sistema e permette di gestire le varie richieste *HTTP*$_G$ provenienti dal *client* attraverso dei ***Logic Adapter*** specifici. In particolare vengono eseguite le seguenti operazioni:

- Ricevere le richieste *HTTP*$_G$ dal *client*;
- Interpretare le richieste scegliendo il *LogicAdapter* corretto;
- Richiedere al *client* eventuale informazioni mancanti per eseguire l'operazione richiesta;
- Determinare con quale microservizio di Imola Informatica interfacciarsi per eseguire l'operazione richiesta;
- Ritornare al *client* una risposta quanto più possibile *human friendly*;

3.2.2 Dipendenze esterne

- **Chatterbot$_G$:** libreria *Python*$_G$ che fornisce le classi per l'interpretazione e le risposte dei messaggi:
  - **Chatbot:** classe concreta con tutte le funzionalità di un *ChatBot*$_G$;
  - **Statement:** classe concreta che rappresenta una frase scritta dall'utente
  - **LogicAdapter:** classe astratta che gestisce e interpreta le frasi scritte dall'utente;
- **View:** classe astratta di *Django*$_G$ che fornisce un API Endpoint per interagire con il *client*;
- **Levenshtein$_G$:** libreria *Python*$_G$ per il calcolo della distanza di Levenshtein$_G$;
- **Requests:** libreria *Python*$_G$ per effettuare chiamate *HTTP*$_G$;
- **Datetime:** libreria *Python*$_G$ per gestire le date;
- **Re:** libreria *Python*$_G$ per utilizzare espressioni regolari;

3.2.3 Diagramma delle classi

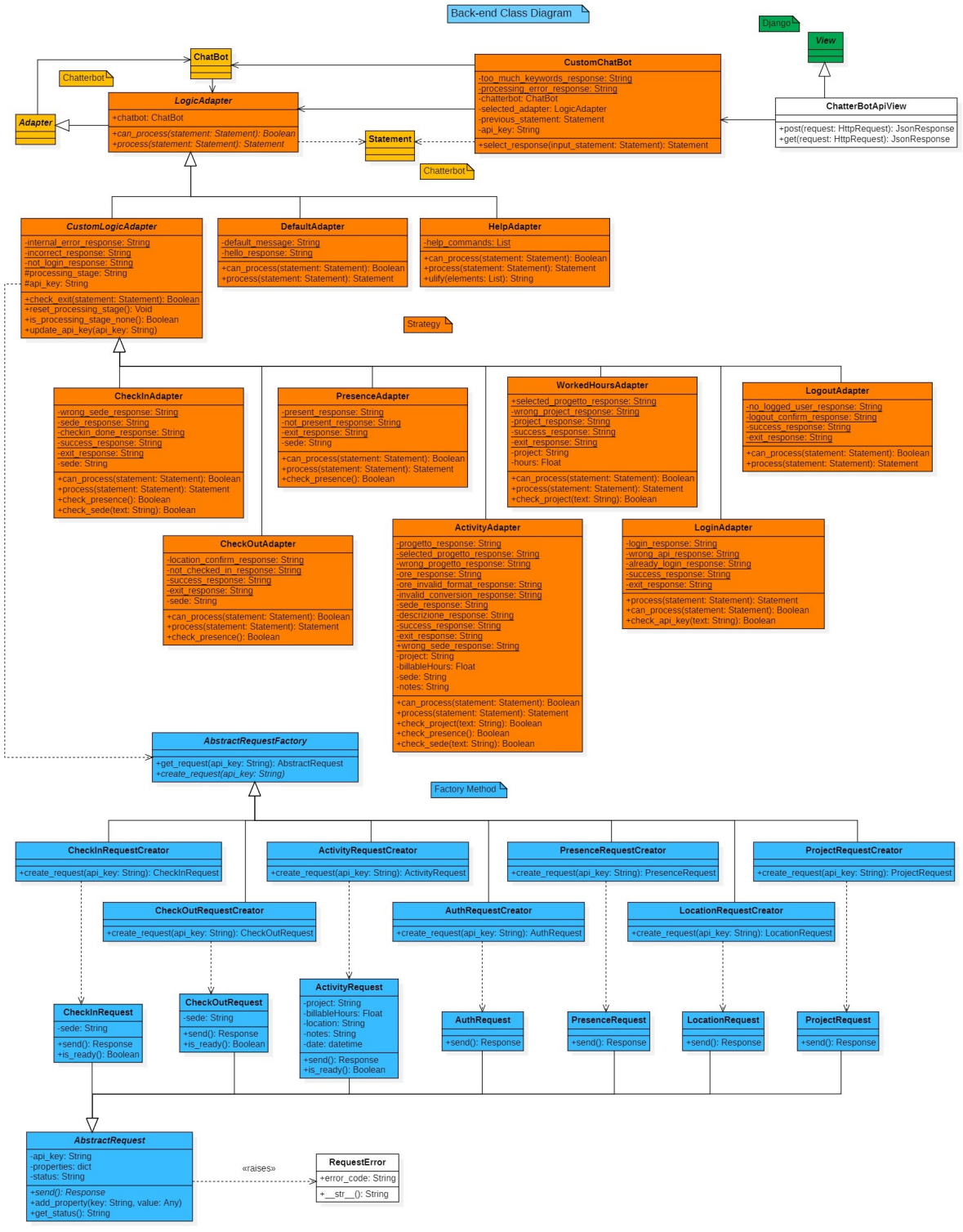

Figura 2: Diagramma delle classi per la parte *back-end*$_G$ dell'applicazione

3.2.3.1 ChatterBotApiView

Rappresenta l'API Endpoint per interagire con il *client*, rispondendo a chiamate 'GET' e 'POST' sull'Endpoint *'/api/chatterbot/'*. Contiene i seguenti metodi:

- ***post(request:HttpRequest)*:** spacchetta la richiesta ricevuta dal *client* e la passa al *CustomChatBot*. Ritorna poi la risposta elaborata da quest'ultimo al *client*;
- ***get(request:HttpRequest)*:** ritorna i dati della conversazione in corso;

3.2.3.2 CustomChatBot

Collega *ChatterbotApiView* con i *Logic Adapter* per poter elaborare una risposta alla richiesta inviata dal *client*. Contiene il seguente metodo:

- ***select_response(input_statement:Statement)*:** seleziona l'$adapter_G$ corretto, ovvero quello in grado di processare l'input inviato dall'utente, e ritorna la risposta migliore fornita dall'$adapter_G$. Nel caso ci sia già un $adapter_G$ selezionato, semplicemente processa l'input e ritorna la risposta corretta.

3.2.3.3 LogicAdapter

Classe astratta di $Chatterbot_G$ che si occupa di gestire e interpretare gli *Statement* ricevuti in input. Definisce due metodi principali che vengono implementati in tutte le sotto classi concrete:

- ***can_process(statement: Statement)*:** controlla che l'$adapter_G$ sia in grado di gestire la richiesta dell'utente;
- ***process(statement: Statement)*:** processa lo *Statement* ricevuto in input e ritorna la risposta corretta;

3.2.3.4 DefaultAdapter

Classe concreta derivata da *LogicAdapter* che si occupa di gestire i messaggi di default del $ChatBot_G$, come il messaggio di benvenuto e il messaggio per quando non è possibile interpretare l'input dell'utente. Questo $adapter_G$ viene selezionato in automatico all'avvio del sistema.

3.2.3.5 HelpAdapter

Classe concreta derivata da *LogicAdapter* che si occupa di gestire e interpretare una richiesta di aiuto, fornendo la lista dei comandi per effettuare le operazioni disponibili.
Contiene inoltre il seguente metodo:

- ***ulify(elements: List)*:** formatta la lista dei comandi in modo tale da renderla comprensibile per il *client*.

3.2.3.6 CustomLogicAdapter

Classe astratta derivata da *LogicAdapter* che si occupa di gestire e interpretare gli input dell'utente e di tener traccia dello stato in cui si trova la conversazione. Contiene i seguenti metodi: Contiene i seguenti metodi:

- ***check_exit(statement:Statement)*:** metodo statico che controlla se è stato richiesto l'annullamento di un'operazione;
- ***reset_processing_stage()*:** esegue un reset dello stato della conversazione;
- ***is_processing_stage_none()*:** controlla se lo stato della conversazione sia nullo;
- ***update_api_key(api_key: String)*:** aggiorna l'$API\ Key_G$ inviata dal *client*;

Ogni sotto classe di *CustomLogicAdapter* necessita per operare della creazione di una *Request*, che rappresenta una richiesta da fare ad uno dei microservizi offerti da Imola Informatica.

3.2.3.7 ActivityAdapter

Classe concreta derivata da *CustomLogicAdapter* che si occupa di gestire e interpretare una richiesta di inserimento di una nuova attività nel sistema EMT_G aziendale. Per fare ciò si serve di un'*ActivityRequest*, che si occupa di fare una chiamata $HTTP_G$ *'POST'* all'Endpoint '/apibot4me.imolinfo.it/v1/projects/{progetto}/activities/me' per inserire l'attività nel sistema EMT_G.
Contiene inoltre i seguenti metodi:

- ***check_project(text:String)*:** controlla se il codice del progetto per cui consuntivare l'attività è valido oppure no, usando *ProjectRequest*;
- ***check_presence()*:** controlla se risulta registrata la presenza in una sede, usando *PresenceRequest*;
- ***check_sede(text:String)*:** controlla la correttezza delle sede in cui è stata svolta l'attività usando *LocationRequest*;

3.2.3.8 CheckInAdapter

Classe concreta derivata da *CustomLogicAdapter* che si occupa di gestire e interpretare una richiesta di registrazione della presenza presso una sede aziendale. Per fare ciò si serve di una *CheckInRequest*, che si occupa di fare una chiamata $HTTP_G$ *'POST'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations/{sede}/presence' per registrare la presenza nel sistema EMT_G aziendale.
Contiene inoltre i seguenti metodi:

- ***check_presence()*:** controlla se risulta già registrata la presenza in una sede, usando *PresenceRequest*;
- ***check_sede(text:String)*:** controlla la correttezza delle sede per cui si vuole fare il *check-in*$_G$ usando *LocationRequest*;

3.2.3.9 CheckOutAdapter

Classe concreta derivata da *CustomLogicAdapter* che si occupa di gestire e interpretare una richiesta di *check-out*$_G$ da una sede aziendale. Per fare ciò si serve di una *CheckOutRequest*, che si occupa di fare una chiamata $HTTP_G$ *'DELETE'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations/{sede}/presence' per togliere la presenza dal sistema EMT_G aziendale.
Contiene inoltre il seguente metodo:

- ***check_presence()*:** controlla se risulta registrata la presenza in una sede, usando *PresenceRequest*;

3.2.3.10 LoginAdapter

Classe concreta derivata da *CustomLogicAdapter* che si occupa di gestire e interpretare una richiesta *login*$_G$. Per fare ciò si serve di una *AuthRequest*, che si occupa di fare una chiamata $HTTP_G$ *'GET'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations' per verificare che l'*API Key*$_G$ inserita dall'utente sia corretta.
Questo controllo viene fatto nel seguente metodo:

- ***check_api_key(text:String)*:** controlla che l'*API Key*$_G$ inserita sia corretta, usando *AuthRequest*;

3.2.3.11 LogoutAdapter

Classe concreta derivata da *CustomLogicAdapter* che si occupa di gestire e interpretare una richiesta *logout*$_G$.

3.2.3.12 WorkedHoursAdapter

Classe concreta derivata da *CustomLogicAdapter* che si occupa di gestire e interpretare una richiesta di visualizzazione delle ore consuntivate per un progetto nella giornata corrente. Per fare ciò si serve di un'*ActivityRequest*, che si occupa di fare una chiamata $HTTP_G$ *'GET'* all'Endpoint '/apibot4me.imolinfo.it/v1/projects/{progetto}/activities/me' per ritornare il numero di ore consuntivate per il progetto inserito.
Contiene inoltre il seguente metodo:

- ***check_project(text:String)*:** controlla se il codice del progetto per cui sapere le ore consuntivate è valido oppure no, usando *ProjectRequest*;

3.2.3.13 PresenceAdapter

Classe concreta derivata da *CustomLogicAdapter* che si occupa di gestire e interpretare una richiesta di visualizzazione dello stato della presenza. Per fare ciò si serve di una *PresenceRequest*, che si occupa di fare una chiamata $HTTP_G$ *'GET'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations/presence/me' per poter ritornare lo stato della presenza in sede.
Questo controllo viene fatto nel seguente metodo:

- ***check_presence()*:** controlla se risulta registrata la presenza in una sede, usando *PresenceRequest*;

3.2.3.14 AbstractRequestFactory

Classe astratta che si occupa di creare una *Request* tramite le sue sotto classi derivate che fungono da *Creator* delle *Request* specifiche. Contiene i seguenti metodi:

- ***get_request(api_key:String)*:** chiama il metodo per costruire una richiesta e la ritorna;
- ***create_request(api_key:String)*:** metodo astratto che costruisce una richiesta, implementato dalle sotto classi;

Per maggior semplicità si elencano qui le sotto classi di *AbstractRequestFactory* incaricate di creare le *Request* specifiche:

- **CheckInRequestCreator:** crea una *CheckInRequest*;
- **CheckOutRequestCreator:** crea una *CheckOutRequest*;
- **ActivityRequestCreator:** crea una *ActivityRequest*;
- **AuthRequestCreator:** crea una *AuthRequest*;
- **PresenceRequestCreator:** crea una *PresenceRequest*;
- **ProjectRequestCreator:** crea una *ProjectRequest*;
- **LocationRequestCreator:** crea una *LocationRequest*;

3.2.3.15 AbstractRequest

Classe astratta che si occupa di gestire le chiamate $HTTP_G$ per una determinata richiesta, interfacciandosi con gli Endpoint di Imola Informatica.
Al suo interno contiene l'*API Key*$_G$ dell'utente e un dizionario dei parametri (*properties*) da passare alla chiamata $HTTP_G$, e lo stato della richiesta.
Contiene inoltre i seguenti metodi:

- ***send()*:** metodo astratto che ritorna la risposta di una chiamata $HTTP_G$. In caso di errore lancia un'eccezione di tipo *RequestError*;
- ***add_property(key:String, value:Any)*:** aggiunge una proprietà al dizionario dei parametri;
- ***get_status()*:** *getter* per lo stato della richiesta;

Tutte le sotto classi di *AbstractRequest* devono ridefinire il metodo *send()*, altrimenti viene lanciata un'eccezione di tipo *RequestError*;

3.2.3.16 RequestError

Classe concreta derivata da *Exception* che rappresenta un'eccezione da lanciare in caso di errore in una *Request*.
Ridefinisce il seguente metodo:

- ***__str__()*:** fa la stampa degli errori direttamente senza accedere ad *.args*

3.2.3.17 CheckInRequest

Classe concreta derivata da *AbstractRequest* che si occupa di gestire una richiesta di *check-in*$_G$. Viene fatta una chiamata *HTTP*$_G$ *'POST'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations/sede/presence' per registrare la presenza nel sistema *EMT*$_G$ aziendale. Contiene inoltre il seguente metodo:

- ***is_ready()*:** controlla che esista il parametro 'sede', necessario per completare la richiesta;

3.2.3.18 CheckOutRequest

Classe concreta derivata da *AbstractRequest* che si occupa di gestire una richiesta di *check-out*$_G$. Viene fatta una chiamata *HTTP*$_G$ *'DELETE'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations/sede/presence' per togliere la presenza dal sistema *EMT*$_G$ aziendale.
Contiene inoltre il seguente metodo:

- ***is_ready()*:** controlla che esista il parametro 'sede', necessario per completare la richiesta;

3.2.3.19 ActivityRequest

Classe concreta derivata da *AbstractRequest* che si occupa di gestire una richiesta di consuntivazione di un'attività. Viene fatta una chiamata *HTTP*$_G$ *'POST'* all'Endpoint '/apibot4me.imolinfo.it/v1/projects/progetto/activiti per inserire un'attività nel sistema *EMT*$_G$, altrimenti viene fatta una chiamata 'GET' all'Endpoint '/apibot4me.imolinfo.it/v1/projects/progetto/activities/me' per ritornare il numero di ore consuntivate per il progetto inserito.
Contiene inoltre il seguente metodo:

- ***is_ready()*:** controlla che esistano i parametri *'sede'*, *'project'* e *'billableHours'* , necessari per completare la richiesta;

3.2.3.20 AuthRequest

Classe concreta derivata da *AbstractRequest* che si occupa di gestire una richiesta di autenticazione. Viene fatta una chiamata *HTTP*$_G$ *'GET'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations' per verificare la correttezza dell'*API Key*$_G$ registrata.

3.2.3.21 PresenceRequest

Classe concreta derivata da *AbstractRequest* che si occupa di gestire una richiesta dello stato della presenza. Viene fatta una chiamata *HTTP*$_G$ *'GET'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations/presence/me' per sapere se si risulta presenti o meno in una sede aziendale.

3.2.3.22 ProjectRequest

Classe concreta derivata da *AbstractRequest* che si occupa di gestire una richiesta dei progetti registrati nel sistema *EMT*$_G$. Viene fatta una chiamata *HTTP*$_G$ *'GET'* all'Endpoint '/apibot4me.imolinfo.it/v1/projects/' per avere l'elenco dei progetti registrati nel sistema.

3.2.3.23 LocationRequest

Classe concreta derivata da *AbstractRequest* che si occupa di gestire una richiesta delle sedi aziendali. Viene fatta una chiamata *HTTP*$_G$ *'GET'* all'Endpoint '/apibot4me.imolinfo.it/v1/locations' per avere l'elenco delle sedi aziendali registrate nel sistema *EMT*$_G$.

3.2.4 Diagramma di sequenza

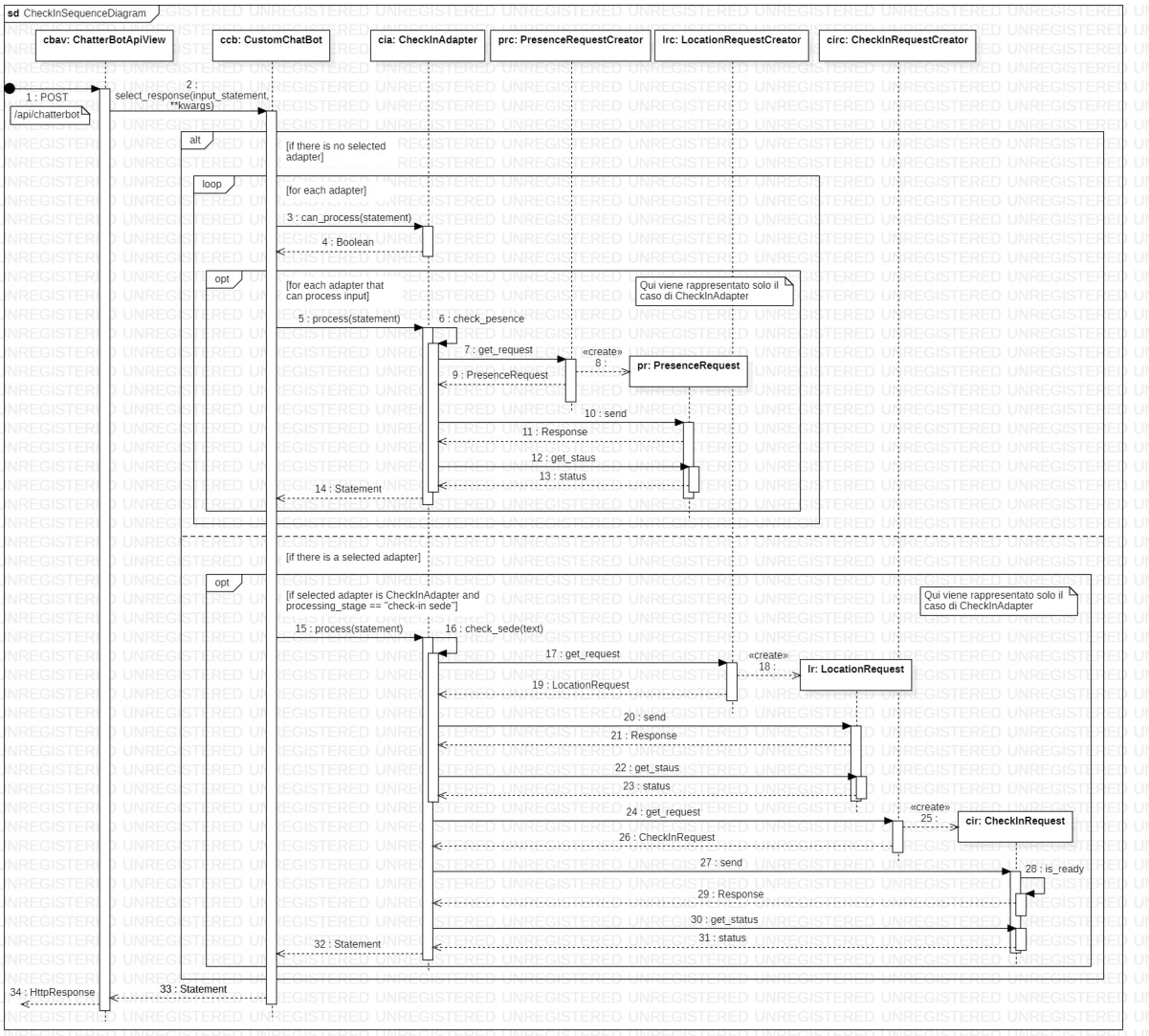

Figura 3: Diagramma di sequenza che mostra il processo di una richiesta di check-in$_G$

Il diagramma descrive gli *step* di una funzionalità offerta dal *ChatBot*$_G$, in questo caso la funzionalità di *check-in*$_G$. Viene reso disponibile un solo diagramma di sequenza, in quanto il processo a cui vanno incontro le diverse funzionalità implementate differisce solo per le chiamate alle richieste corrispondenti. Il funzionamento di una qualsiasi funzionalità è dunque il seguente:

1. Il *client* effettua una chiamata POST, nel cui *body* è contentua il messaggio dell'utente, all'*Endpoint*$_G$ "/api/chatterbot/", come meglio descritto nella sezione §3.3.1.2.

2. La *view* riceve la chiamata, riconosce il messaggio dell'utente e lo invia al *ChatBot*$_G$.

3. Il *ChatBot*$_G$ controlla il suo stato interno e procede secondo i seguenti casi:

   - Nel caso in cui non ci sia alcun *Logic Adapter* selezionato:

(a) Effettua una chiamata al metodo can_process() per ogni singolo *adapter*$_G$ e per verificare quali *adapter*$_G$ possono processare la richiesta.

(b) Se la chiamata ritorna True, chiama il metodo process() in cui, se l'adapter lo prevede, viene allocato una *Request* discendente da *AbstractRequest* tramite il *Creator* corrispondente. Prima della creazione della richiesta l'*adapter*$_G$ controlla inoltre che siano presenti tutti i parametri necessari.

(c) Nel caso venga utilizzata una *Request*, essa provvede ad effettuare la richiesta all'*Endpoint*$_G$ corretto e a ritornare la risposta della chiamata HTTP$_G$, utilizzata poi dall'*Adapter*.

(d) Se la chiamata al metodo *can_process()* ritorna False, invece, non viene effettuata alcuna operazione.

(e) Una volta prodotte le risposte dai singoli *adapter*$_G$, viene scelta quella con l'indice di confidenza più alto, nel caso in cui più risposte abbiano lo stesso valore, allora verrà informato l'utente della mancata comprensione della richiesta.

(f) Se l'*adapter*$_G$ ha bisogno di altri input intermedi da parte dell'utente per processare correttamente la richiesta, allora viene selezionato, in modo tale da poter proseguire alla prossima chiamata del *client*.

- Nel caso in cui ci sia un *Logic Adapter* selezionato:

(a) Chiama il metodo process() del *Logic Adapter* selezionato in precedenza, in cui viene continuata la procedura necessaria per il corretto completamento della richiesta.

(b) Come nel primo caso, l'*adapter*$_G$ rimane selezionato se ha bisogno di altri input intermedi da parte dell'utente, mentre se ha completato la richiesta viene deselezionato.

4. Il *ChatBot*$_G$ ritorna la risposta alla *view*.
5. La *view serializza* la risposta ricevuta e la rispedisce al *client* sotto forma di *JsonResponse*.

3.2.5 Design pattern

Per la parte *back-end*$_G$ dell'applicazione software sono stati utilizzati due *design pattern*$_G$: Strategy e Factory Method.

- **Strategy**: Viene utilizzato nella parte di selezione della risposta da parte del *ChatBot*$_G$.
  I vari *Logic Adapter* vengono visti come algoritmi diversi che hanno lo stesso scopo, ovvero quello di produrre una risposta sotto forma di *Statement*. Il *ChatBot*$_G$ inoltre, cambia la tipologia di *Logic Adapter* a run-time in base alla richiesta dell'utente.
  Il pattern viene indicato nel diagramma delle classi con il colore arancione.
- **Factory Method**: Viene utilizzato per creare gli oggetti che effettuano le richieste alle *API*$_G$ di Imola Informatica, in modo tale da applicare il *Single Responsibility Principle* e l'*Open-Closed Principle*, rendendo così il codice più facilmente estendibile.
  Il pattern viene indicato nel diagramma delle classi con il colore azzurro.

3.3 REST API$_G$

3.3.1 API$_G$ ChatBot$_G$

L'indirizzo usato come base per gli API Endpoint del *ChatBot*$_G$ è il seguente:

https://skynetbot.onrender.com/

3.3.1.1 Inizio conversazione

All'apertura del *ChatBot*$_G$ il *client* elabora la seguente richiesta, che in caso di successo ritorna un messaggio di benvenuto:

API Endpoint
/api/chatterbot/

Chiamata HTTP$_G$
'GET'

Parametri
Nessuno

Headers

- **Content-type:** "application/json"
- **Authorization:** api_key (facoltativa)

Risposta

Status Code	Body (JSON)	Descrizione
200	{"text": string}	Il *server* ritorna al *client* un messaggio di benvenuto

3.3.1.2 Conversazione

Successivamente al messaggio di benvenuto, il *client* riceve i messaggi dal *server* utilizzando il seguente Endpoint, per poter interpretare i messaggi e fornire le risposte corrette:

API Endpoint
/api/chatterbot/

Chiamata HTTP$_G$
'POST'

Parametri

Tipo	Body (JSON)	Descrizione
JSON Object	{"text": string, "api_key": string}	L'attributo "text" contiene il messaggio inviato dall'utente. L'attributo "api_key invece contiene l'API Key$_G$ dell'utente

Headers

- **Content-type:** "application/json"
- **Authorization:** api_key

Risposta

Status Code	Body (JSON)	Descrizione
200	{"text": string, "in_response_to": string, "created_at": string}	Il messaggio viene interpretato correttamente. e il *body* contiene la risposta da ritornare al *client*
400	{"text": string}	Non è stato specificato il campo "text" quindi non è possibile elaborare il messaggio.

3.3.2 API_G Imola Informatica

Il proponente Imola Informatica ha fornito delle *REST* API_G per poter effettuare le operazioni richieste tramite il $ChatBot_G$. Tali API_G sono descritte in dettaglio nello $Swagger_G$ *UI* fornito dall'azienda, disponibile al seguente link: https://apibot4me.imolinfo.it

4 Setup

4.1 Requisiti di sistema

Per il deploy del software è necessario aver installato i seguenti linguaggi/librerie:

- Python 3.7.9
- Node.js
- npm

4.2 Deploy

Per il *deploy* in locale dell'applicativo tramite linea di comando è necessario effettuare i seguenti passi:

1. Aprire la cartella contenente i file del progetto *SkyNetChatbot*
2. Installare le dipendenze tramite i comandi

   ```
   npm install -g yarn
   yarn install
   ```

3. Far partire il $ChatBot_G$ con il comando

   ```
   yarn run dev (per Mac e Linux)
   yarn run devwin (per Windows)
   ```

Per il deploy finale dell'applicativo sul web è stato utilizzato $Render_G$, che permette di accedere alla *Web* App_G sviluppata tramite l'indirizzo https://skynetbot.onrender.com.

4.3 Testing

I test di unità del software sono stati sviluppati utilizzando il modulo **unittest** contenuto in $Django_G$. Per eseguire tutti i test sviluppati per il software è necessario effettuare i seguenti passi da linea di comando:

1. Aprire la cartella *SkyNetChatbot* contenuta in *SkyNetChatbotVirtualEnv*
2. Far partire i test con il comando

   ```
   python3 manage.py test (per Mac e Linux)
   py -3.7 manage.py test (per Windows)
   ```

Una volta che i test verranno effettuati, nel log_G verrà mostrato l'output contenente:

- Il numero totale di test effettuati
- Il tempo impiegato per effettuare i test
- Il risultato dei test: se nessuno fallisce compare la scritta *"OK"*, altrimenti viene indicato il numero e il motivo dei fallimenti