Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea Magistrale in Informatica

Un approccio innovativo all'addestramento di chatbot intelligenti

Tesi di Laurea Magistrale

Relatore

Prof.ssa Ombretta Gaggi

Laureando

Michele Tagliabue
Matricola 1206966

Anno Accademico 2020-2021

Michele Tagliabue: *Un approccio innovativo all'addestramento di chatbot intelligenti,* Tesi di Laurea Magistrale, © 22 Aprile 2021.

“Chi un giorno vuole imparare a volare, deve prima di tutto imparare ad alzarsi e andare e camminare e arrampicarsi e danzare: volare non si impara volando.”

Friedrich Nietzsche

Abstract

La gestione delle comunicazioni con i clienti rappresenta un processo determinante per il successo di un'azienda. I processi di *lead generation* e *customer care* richiedono una gestione della messaggistica aziendale assai efficiente. Molto spesso però la maggior parte delle richieste di un utente si assomigliano tra loro, rendendo possibile la creazione di software in grado di fornire risposte automatiche che riducono il carico di lavoro agli operatori umani. Grazie all'intelligenza artificiale, questi strumenti si sono evoluti in *chatbot* in grado di interpretare in modo più accurato l'input dell'utente, oltre che a sostenere conversazioni complesse.

Il processo di creazione di un *chatbot* è molto complesso, ma ultimamente sono nate delle piattaforme che si prefiggono lo scopo di renderlo alla portata di tutti, targetizzando soprattutto persone senza competenze informatiche. Tuttavia la maggior parte di loro non permette di applicare tecniche di NLP (*Natural Language Processing*), impedendo quindi la creazione di *chatbot* realmente intelligenti.

Lo scopo di questa tesi è stato realizzare un'applicazione che assista un utente inesperto nel processo di design e creazione di un *chatbot*. Questo viene poi installato su una piattaforma esterna, come *Amazon Lex*, che consente l'applicazione di tecniche di NLP e soprattutto NLU (*Natural Language Understanding*) in grado di migliorare la capacità del bot di comprendere le richieste dell'interlocutore. L'applicazione realizzata non richiede competenze informatiche e consente di costruire risposte anche complesse tramite una procedura guidata ed un editor interamente visuale.

Ringraziamenti

Non nascondo che gli ultimi due anni della mia vita, nei quali è caduto il periodo della stesura di questa tesi, non sono stati per niente facili. Ho avuto molti momenti bui, di blackout completo, commistione tra crollo emotivo e fisico, che mai avrei pensato mi potessero capitare. Mi sono sentito perso, come un pesce d'acquario che all'improvviso si ritrova a nuotare nell'oceano, solo ad affrontare delle difficoltà che percepivo grandi come montagne. Ho pensato più volte di mollare, di abbandonare il percorso di studi che oggi, *22 Aprile 2021*, sto finalmente portando a compimento.

In tutto questo ci sono state delle persone che, magari anche inconsciamente, hanno saputo aiutarmi. Ora intendo ringraziarle, una ad una, qui, perché se lo meritano, e perché è il minimo gesto di riconoscenza che possa fare.

Inizio con mia madre, Antonella, santa donna (amen), che ha saputo supportarmi sempre, in ogni momento, a prescindere. Proseguo con mio padre, Mauro, di cui non posso far altro che ammirare la sua caparbietà, il suo istinto a non mollare mai e la sua capacità di influenzare anche le altre persone a non farlo.

Continuo con il ringraziare tutti i miei amici, ma in particolare tre di loro, per esserci **sempre** stati al momento del bisogno, senza **mai** chiedere nulla in cambio e per avermi fatto presente, quando necessario, il mio valore:

- *Paolo Ceccon*, per essere riuscito a starmi vicino pur trovandosi a più di ottomila kilometri di distanza. *Paul, per quanto mi riguarda sei il fratello che non ho mai avuto*;
- *Marco Cesarato*, il mio *hacker di fiducia*, che non si è mai tirato indietro quando è stata ora di aiutarmi, che è riuscito a ricordarmi con poche ed efficaci parole il mio valore, che si è ritrovato ad essere (suo malgrado, LOL) il mio psicologo. *Cesa, sappi che sei una persona che stimo, sia come programmatore ma ancora di più come amico*;
- *Daniele Penazzo*, per avermi aiutato nella stesura e nella correzione della tesi, rubando del tempo al suo lavoro. *Penaz, spero un giorno di poter ricambiare gli innumerevoli favori ricevuti, perché te lo meriti!*

Infine un pensiero non può che andare a due persone che purtroppo non ci sono più, ma i cui insegnamenti vivranno in me per sempre. La prima di queste è la prof.ssa di educazione fisica delle scuole medie *Daniela Zuin*, che è riuscita ad insegnarmi il seguente mantra: "ogni caduta è un'occasione in più per imparare ad alzarsi".
Vorrei ora concludere ricordando *Sergio Rosati*, canoista ma soprattutto maestro di vita. Me lo sono immaginato molte volte, soprattutto nei momenti più bui, lì davanti a me, a ribadirmi, con tono molto pacato: *"Tagliabue, TIRA FUORI I COGLIONI!!!!"*.

Padova, 22 Aprile 2021 Michele Tagliabue

Indice

Elenco delle figure

Capitolo 1

Introduzione

I *chatbot* sono un argomento sempre più attuale, soprattutto nel mondo enterprise. Un sondaggio del 2018 condotto da Oracle che ha coinvolto circa 800 suoi partner commerciali ha evidenziato come l'80% di loro ha programmato di implementare una soluzione *chatbot* per la gestione del servizio clienti, mentre il 36% lo ha già fatto [15].

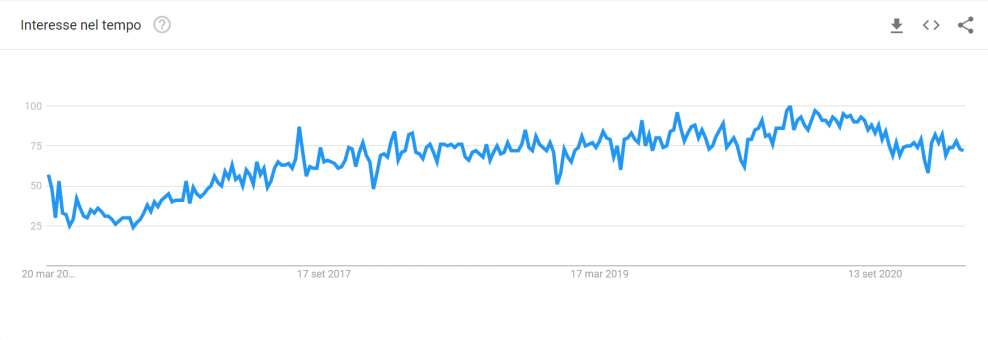


Figura 1.1: Andamento dell'interesse rilevato da Google sulla query di ricerca "chatbot" da marzo 2016 a marzo 2021

Inoltre anche analizzando la frequenza delle ricerche su Google con query "chatbot", illustrata in figura 1.1, si nota come l'interesse verso questo argomento sia molto in voga.

A livello implementativo, un *chatbot* ha una struttura molto complessa, in quanto richiede, tra le altre cose, la comprensione dell'input dell'utente oltre che la definizione di una risposta per ogni possibile input. Ecco che la realizzazione di un *chatbot* può risultare molto lunga e complessa, decisamente fuori portata dell'utente medio, ma anzi molto spesso appannaggio di aziende o team multidisciplinari, alla quale partecipano esperti informatici (incaricati dell'implementazione vera e propria), linguisti (il cui compito è analizzare e definire le modalità di comunicazione con l'utente) e commerciali (in grado di fornire le risposte rilevanti nel dominio).

In un'ottica di "democratizzazione" del processo di creazione di un *chatbot* sono nate delle piattaforme che si prefiggono di renderlo meno costoso in termini di tempo e risorse necessarie. Il mercato risulta molto frammentato, anche se le offerte presenti si possono raggruppare in due categorie: piattaforme *business-oriented*, aventi come target le aziende che mirano a creare *chatbot* usabili nel minor tempo possibile, e piattaforme *generaliste*, che puntano più sulla completezza delle funzionalità offerte che sulla facilità di realizzazione.

L'obiettivo della tesi, proposta dall'azienda Primo Round s.r.l. è stato creare una piattaforma di progettazione ed addestramento per *chatbot* intelligenti che non richieda competenze informatiche specifiche per essere utilizzata. L'idea nasce dal fatto che, come verrà approfondito nel capitolo 3, le piattaforme *business-oriented* non dispongono di motori di NLP ed NLU efficaci, quindi permettono di realizzare dei *chatbot* molto basilari. D'altro canto, le piattaforme *generaliste* sono molto potenti ma molto più complicate da usare: manca quindi una via di mezzo, ovvero un modo semplice per permettere a tutti di sfruttare al meglio le piattaforme generaliste.

Il progetto di tesi è iniziato a settembre 2020 con uno stage presso l'azienda Primo Round s.r.l. che si è protratto per tre mesi. Il lavoro è stato suddiviso in:

1. **Analisi dello stato dell'arte**: test delle principali piattaforme di addestramento *business-oriented* e *generaliste* per capirne pregi, difetti e modalità di funzionamento;
2. **Esame di un mockup grafico** realizzato con *Adobe XD* dall'azienda;
3. **Progettazione ed implementazione** della soluzione.

Capitolo 2

Interazione uomo-macchina

2.1 Concetti principali

Nel contesto dell'Human Computer Interaction, con il termine interazione ci si riferisce alla comunicazione tra l'utente e il sistema (computer). Questi due soggetti hanno caratteristiche e modalità di operare molto differenti, pertanto è necessario uno strumento che faccia "da ponte", ovvero che traduca le intenzioni dell'utente in azioni comprensibili al sistema: l'interfaccia. Affinché l'interazione avvenga con successo, il sistema deve risultare il più efficace possibile nel permettere all'utente di realizzare i propri obiettivi. Ecco che esistono dei modelli per analizzare la qualità dell'interazione tramite un'interfaccia, in modo da valutare eventuali pregi e/o carenze: il più famoso è il modello di Norman.

2.1.1 Modello di Norman

2.1.1.1 Termini fondamentali

Secondo il modello di Norman [7], gli elementi che caratterizzano l'interazione uomo-macchina sono:

- **Dominio**, ovvero l'insieme di concetti, competenze e conoscenze che caratterizzano l'area applicativa oggetto dello studio;
- **Compito (task)**, l'insieme di operazioni che agiscono sugli elementi del dominio. In altre parole, rappresenta l'insieme di azioni che l'utente deve svolgere per portare a compimento il suo obiettivo;
- **Obiettivo**, cioè l'output atteso di un compito, ovvero quello che l'utente vuole fare;
- **Intenzione**, sequenza di azioni generiche che portano al raggiungimento dell'obiettivo.

Nell'analisi dell'interazione uomo macchina possiamo distinguere due attori, il sistema e l'utente, ciascuno dei quali possiede un proprio stato e "parla" un proprio linguaggio (rispettivamente core language e task language).

2.1.1.2 Funzionamento del modello

Secondo il modello di Norman (sintetizzato in figura 2.1), l'utente per provare a raggiungere l'obiettivo realizza un piano d'azione, che poi attua interagendo con l'interfaccia. Al termine

dell'esecuzione di tale piano, l'utente ne osserva gli effetti e decide le eventuali azioni successive da compiere.

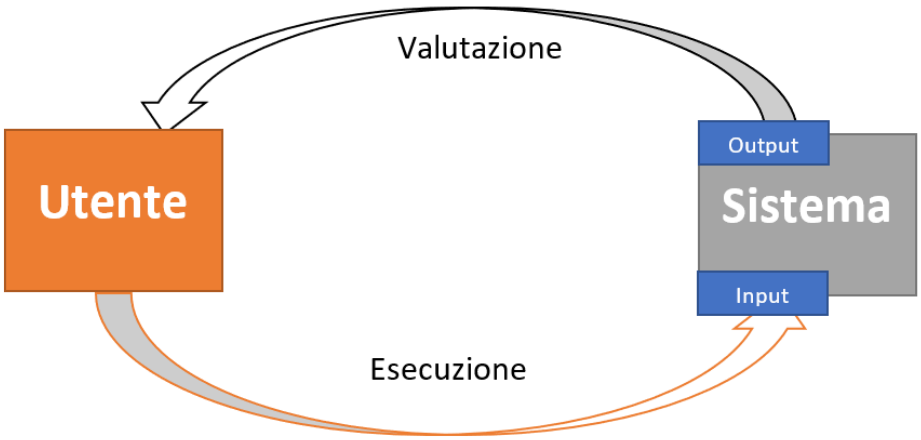


Figura 2.1: Modello di Norman

Le fasi di esecuzione e valutazione possono essere ulteriormente scomposte in:

- **Esecuzione**:
  1. Stabilire l'obiettivo, cioè definire *COSA* si vuole fare.
  2. Pianificare l'intenzione, che corrisponde a pensare COME si vuole raggiungere l'obiettivo.
  3. Specificare la lista di azioni da eseguire.
  4. Eseguire le azioni
- **Valutazione**:
  1. Percepire lo stato del sistema.
  2. Interpretare lo stato del sistema.
  3. Valutare lo stato del sistema rispetto le attese.
  4. Eseguire le azioni

Secondo Norman ogni soggetto (utente e sistema) ha il proprio modo di esprimere concetti rilevanti per il contesto dell'applicazione. Questi modi sono chiamati *linguaggi* e si distinguono [7]:

- *core language*: permette di descrivere l'applicativo (inteso come dominio delle azioni possibili e task da svolgere) dal "punto di vista del sistema".
- *task language*: permette di descrivere il sistema dal punto di vista dell'utente.

Quando l'utente svolge una delle fasi di esecuzione o valutazione "parla" il task language, mentre il sistema risponde usando il core language. Un fallimento possibile dell'interazione uomo-macchina è l'errata traduzione tra i due linguaggi, che presenta due "punti critici":

- *Golfo di Esecuzione*, ovvero la differenza tra le azioni formulate dall'utente e quelle ammesse dal sistema;
- *Golfo di Valutazione*, ovvero la differenza tra lo stato in cui si trova il sistema e quello percepito dall'utente.

Come emerge dalle affermazioni precedenti, il modello di Norman effettua un'analisi lato utente, ignorando completamente le modalità di comunicazione del sistema con l'interfaccia. Per questo motivo è stata proposta un'estensione di tale modello, chiamata *Interaction Framework*.

2.1.2 Interaction framework

2.1.2.1 Descrizione del modello

L'interaction framework [7] è essenzialmente un'ulteriore scomposizione del modello di Norman che rende esplicito il ruolo dell'interfaccia utente. L'interazione viene definita da quattro componenti principali, ciascuna con il proprio linguaggio: utente, sistema, input e output. Queste ultime due rappresentano l'interfaccia.
Durante la fase di esecuzione, l'utente non interagisce più direttamente con il sistema, ma solo con la componente input: il *task language* viene quindi convertito in *input language*, il quale poi a sua volta verrà convertito in *core language* interpretato dal sistema. In fase di valutazione lo stato del sistema verrà convertito proiettato nello spazio di output (cioè descritto usando l'*output language*), che è quello che verrà osservato dall'utente e utilizzato per la valutazione.

2.1.2.2 Analisi degli errori

L'*Interaction Framework* ammette errori di interazione causati da uno o più incongruenze durante i seguenti mapping:

- *Articulation*, ovvero la traduzione dell'intenzione dell'utente in input language;
- *Performance*, cioè l'interpretazione da parte del sistema delle azioni dell'utente (descritte con l'input language) e il conseguente adeguamento del suo stato interno (compimento dell'elaborazione);
- *Presentation*, la traduzione dello stato del sistema nello spazio definito dall'output language;
- *Observation*, cioè la percezione da parte dell'utente dello stato del sistema output del mapping presentation.

Una rappresentazione schematica dell'interazione è presente in figura 2.2.
La fase di *articulation* è tanto più semplice quanto più diretto è il mapping tra gli attributi psicologici specifici del dominio del task e l'input language. In altre parole, più l'input language permette di descrivere facilmente l'intenzione dell'utente, minore sarà il rischio che si verifichino errori durante questa fase. Volendo illustrare il concetto con un esempio, pensiamo ad una macchina radiocomandata che può muoversi in avanti, indietro, destra e sinistra. Ipotizziamo che vengano forniti due radiocomandi: uno con i bottoni disposti a croce e l'altro con i bottoni disposti in linea. Per l'utente sarà molto più immediato impartire comandi con il primo radiocomando, in quanto la disposizione dei tasti è più coerente con la rappresentazione del comando da impartire.
Durante la fase di *performance*, quello che conta è la completezza dell'output language nello

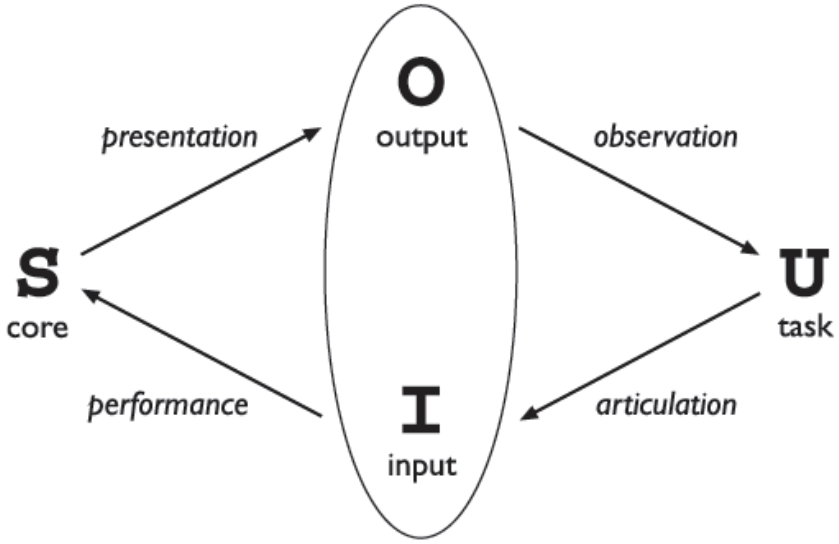


Figura 2.2: Interaction framework. Fonte: https://bit.ly/3uOcYVc

mappare le funzionalità del sistema. Tornando all'esempio della macchina radiocomandata, un fallimento nella fase di performance potrebbe essere rappresentato da un radiocomando con solo due tasti (avanti e indietro), che renderebbe impossibile sterzare la macchina (funzionalità che invece la macchina possiede).

2.2 Storia dell'interazione uomo macchina: dal mouse alle interfacce funzionali

2.2.1 Dal sistema batch alla GUI

2.2.1.1 Sistema batch

Agli albori dell'informatica, il computer venne concepito come una macchina in grado di eseguire calcoli complessi, senza alcuna separazione tra dati e istruzioni: erano macchine *single-purpose*. L'esempio più celebre di un dispositivo di questo tipo è sicuramente Colossus, realizzato per decrittografare i messaggi dei nazisti durante la seconda guerra mondiale.
Con l'avanzare del progresso tecnologico si giunse al concetto di programma, quindi all'idea che un calcolatore potesse svolgere compiti diversi in base a parametri forniti in input dall'utente. Spesso il programma era in forma di schede perforate che venivano inserite nel calcolatore. Una volta terminata l'elaborazione veniva fornito un output, che poteva essere ad esempio scritto su un nastro magnetico, stampato o perforato su un'apposita scheda.
In questo stadio di evoluzione l'uso del computer era riservato a professionisti del settore quindi non vi era la necessità di un'interfaccia utente.

2.2.1.2 CLI

Lo step evolutivo successivo fu sancito con l'avvento delle interfacce a riga di comando. I computer non erano ancora "personal": al contrario, erano mainframe, molto spesso situati in

stanze apposite.
All'epoca era presente un dispositivo chiamato teleprinter (o *TeleTYpewriter*, *TTY*) dotato di tastiera e stampante in grado di trasmettere e ricevere messaggi testuali tramite la rete telegrafica. Nello specifico, quando si voleva trasmettere un messaggio si utilizzava la tastiera e tutti i messaggi ricevuti venivano stampati (le ultime versioni di questo sistema al posto della stampante adottavano un monitor a fosfori verdi per l'output). Connettendo un teleprinter ad un mainframe si ottennero delle vere e proprie postazioni remote: i comandi venivano impartiti tramite tastiera e l'output ricevuto sotto forma di messaggio testuale, inizialmente stampato e successivamente visualizzato a video (questi ultimi chiamati glass tty).
L'introduzione del monitor permise di ridurre la latenza di interazione e, volendo analizzare anche l'impatto economico/ambientale, di ridurre enormemente l'utilizzo di carta ed inchiostro, riducendo quindi il costo di esecuzione di un singolo programma. I principali pregi di questa soluzione sono il basso overhead in termini di risorse necessarie al suo funzionamento e la rapidità di utilizzo per un utente avanzato: conoscendolo, si può impartire direttamente un comando specifico (al contrario di quanto avverrà/avviene con le GUI). I difetti sono sicuramente la bassa user-friendliness e l'elevata curva di apprendimento: banalmente è necessario apprendere i comandi da impartire. Inoltre le modalità di output sono limitate: anche nei moderni sistemi operativi, tramite un emulatore di terminale, non possono essere visualizzate immagini o elementi grafici, perché l'interfaccia è incentrata sul testo.
Al contrario delle schede perforate, le interfacce a riga di comando sono sopravvissute fino ai giorni nostri. Nei moderni sistemi operativi infatti è quasi sempre presente un emulatore di terminale (es: "PowerShell" per Windows) che permette di impartire comandi ed eseguire applicazioni in modalità testuale. Inoltre il terminale è ancora molto utilizzato per la manutenzione di dispositivi da remoto quali server, router e tutti quegli apparati appartenenti all'IoT che prendono sempre più piede.

2.2.1.3 GUI - Graphical User Interfaces

2.2.1.3.1 La visione di Englebart Le origini dell'interfaccia grafica sono da ricondurre al genio di Douglass Englebart. Egli, fortemente ispirato dal progetto *Memex* di *Vannevar Bush*, fu il primo a realizzare un computer il cui scopo primario non era fare calcoli, ma diventare un assistente e "potenziatore" delle capacità umane e che facilitasse l'interazione e la collaborazione tra individui. Il suo pensiero fu sintetizzato nel paper *"Augmenting the Human Intellect: A conceptual framework."* del 1962, in cui secondo me emblematica è la seguente frase: *"We see the quickest gains emerging from (1) giving the human the minute-by-minute services of a digital computer equipped with computer-driven cathode-ray-tube display, and (2) developing the new methods of thinking and working that allow the human to capitalize upon the computer's help."* [2].

Con i fondi di DARPA e NASA, Englebart riesce a concretizzare le sue idee dando vita all'*NLS*, acronimo di *oNLine System* (mostrato in figura 2.3): un intero sistema che prometteva di rivoluzionare l'approccio all'utilizzo del computer, fornendo un'interfaccia grafica basata su finestre ed elementi point and click, un sistema di teleconferenza e di collaborazione online real-time (in stile google documenti), e delle nuove periferiche per l'interazione con il computer, tra le quali il mouse.
NLS venne presentato nel 1968 in una dimostrazione di 90 minuti (Mother of all Demos), durante la quale Englebart fece vedere un quantitativo di novità incommensurabili: finestre, link dinamici, mouse ed interazione point and click, videoconferenze, live editing in stile google documenti, online collaboration (due mouse sullo schermo, uno suo e uno dell'utente remoto) e altre cose. La presentazione però fu talmente tanto rivoluzionaria da sembrare

Figura 2.3: Foto dell'oNLine System ideato da Englebart. Fonte: https://www.darpa.mil/about-us/timeline/nls

fantascienza, quindi le novità tecnologiche presentate non ebbero un forte ed immediato impatto sull'informatica dell'epoca.

2.2.1.3.2 Il primo esempio di interfaccia moderna Fu solo qualche anno più tardi che alcune delle novità di Englebart vennero implementate in un prodotto costruito in massa. Nel 1973 venne rilasciato lo Xerox Alto, un computer dotato di monitor, mouse, tastiera ed un'interfaccia visuale (GUI) a finestre, fortemente ispirato all'NLS, anche perché molti dei colleghi di Englebart tra il 1968 e il 1973 lasciarono l'ARC e vennero assunti dalla stessa Xerox. Lo Xerox Alto portò il primo esempio di interfaccia *WIMP*, acronimo di *Windows, Icon, Menu and Pointing Device*. Questo device però venne prodotto in pochissimi esemplari, riservati tutti al mondo della ricerca scientifica, ma qualche anno più tardi questa tecnologia (e visione) venne portata sul mercato con lo Xerox Star. Tale dispositivo, seppur molto costoso (l'equivalente di circa 40 mila dollari odierni), disponeva di un'interfaccia WIMP con un programma di videoscrittura *WYSIWYG* (acronimo di *What You See Is What You Get*) chiamato *BravoX*, di connettività "completa" (era dotato di porta Ethernet per la partecipazione a reti di calcolatori) e di periferiche all'avanguardia, come un mouse a due tasti e un monitor da 17 pollici [6].
Nonostante fosse all'avanguardia sotto tutti i punti di vista, commercialmente si rivelò un fallimento: probabilmente era troppo in anticipo nei tempi e i consumatori non riuscirono a percepirne il valore e quindi a giustificare l'elevatissimo costo di acquisto. A supporto di questa tesi c'è anche il fallimento dell'Apple Lisa, sistema rilasciato nel 1983 e di concezione molto simile a quelle dello Xerox Star, sia a livello di interfaccia grafica che di posizionamento nel mercato in termini di prezzo di vendita.

Per l'"utente medio" la vera rivoluzione arrivò solo nel 1984 con il lancio dell'*Apple Macintosh* (mostrato in figura 2.4) che, rispetto al *Lisa* era meno potente ma allo stesso tempo molto più economico. Ecco che l'utente medio non era in grado di giustificare l'esborso economico notevolmente superiore necessario a comprare uno *Xerox Star* o un *Lisa*, quando con una frazione del prezzo poteva ottenere un prodotto ai suoi occhi non tanto dissimile.

Figura 2.4: Foto dell'Apple Macintosh. Fonte: https://bit.ly/2ZMAX8T

2.2.1.3.3 Interfacce post-WIMP Il tipo di interazione rivoluzionario introdotto da Xerox e Apple nei primi anni '80 ha definito paradigmi di design e convenzioni che sono state mantenute fino all'epoca odierna. Infatti, anche nei moderni sistemi operativi (come Windows 10 e Mac OS X) le applicazioni girano su finestre, nelle quali sono presenti icone e menù cliccabili tramite l'uso del mouse.
Tra la fine degli anni '90 e i primi anni 2000, grazie ai progressi fatti in ambito di miniaturizzazione della capacità computazionale, sono nate nuove categorie di dispositivi, il cui form factor era notevolmente differente rispetto a quanto visto fino ad ora. Un esempio sono stati i *computer palmari* (*PDA*), dispositivi che, come suggerisce il nome, sono stati pensati per stare ed essere utilizzati sul palmo di una mano. Questo comporta una notevole differenza rispetto ai computer tradizionali in termini di modalità di interazione. Ad esempio, la ridotta dimensione dello schermo rende difficoltosa la presenza di più contenuti (finestre) contemporaneamente. Inoltre, dovendo essere utilizzati in mobilità, questi dispositivi non sono dotati di mouse, bensì di schermo tattile. Infine molti di essi erano sprovvisti di tastiera fisica.

Figura 2.5: Esempio di PDA: Palm Tungsten T3. Fonte: https://amzn.to/2ZISBud

La sfida innescata nel rendere quanto più fruibili device di questo tipo ha portato alla sperimentazione di nuove modalità di interazione, nuove interfacce che prendono il nome di *post-WIMP* dove vi è la mancanza di almeno uno degli elementi caratterizzanti, quali finestre, menu e widget 2D in generale [5]. Esempi possono essere la navigazione tramite gesture (presente già ai tempi dell'*Apple Newton*) o l'utilizzo del riconoscimento vocale, sia come metodo di immissione di testo che come strumento per impartire comandi.

2.2.1.4 CUI - Conversational User Interfaces

In realtà il riconoscimento vocale è una tecnologia che si è iniziata a sviluppare già qualche anno prima.
Ad esempio, nel 1952 un gruppo di ricercatori dei *Bell Telephone Labs* pubblicarono un paper [1] in cui veniva illustrato un dispositivo completamente analogico (chiamato *Audrey*) in grado di riconoscere le cifre dettate da una specifica persona. Tale dispositivo faceva uso di tecniche di analisi del suono volte ad isolare le singole frequenze sonore caratteristiche di ogni cifra. Ovviamente ogni individuo ha una voce/timbro vocale differente, quindi il dispositivo era "personale", ovvero doveva essere tarato sul modello vocale di un singolo soggetto. Una volta svolta questa operazione permetteva di raggiungere un'accuratezza del riconoscimento oscillante tra il 97 e il 99%. La necessità di essere tarato su ogni interlocutore però costituiva anche il suo difetto principale: se è vero che poteva essere utilizzato, ad esempio, da un centralinista per digitare un numero di telefono, è anche vero che, richiedendo pause di 350ms tra una cifra e l'altra, l'inserimento risultava molto più lento che utilizzando la tastiera. [4]
Esattamente dieci anni più tardi IBM presentò il *Shoebox*, un apparecchio in grado di riconoscere 16 vocaboli in lingua inglese: le cifre decimali e sei comandi di operazioni aritmetiche (come "plus", "minus", "total"). In questo modo poteva poi comandare una calcolatrice, che restituiva il risultato dell'operazione artimetica su un foglio di carta. [3]
Nel 1971 la *DARPA* finanziò un programma di ricerca quinquennale chiamato *DARPA Speech Understanding Research* che permise la nascita di *HARPY*, un sistema in grado di riconoscere non singole parole bensì intere frasi costruite con un vocabolario di 1011 parole. Questo fu il primo sistema ad usare in modo efficace un modello di linguaggio (*language model*) per aumentare la precisione del riconoscimento scartando frasi prive di significato [10] utilizzando una tecnica di *beam search*. [4]
Uno step evolutivo successivo si ebbe con l'applicazione degli *Hidden Markov Model* per la realizzazione del modello linguistico. Questo strumento applica un ragionamento di tipo probabilistico per modellare la probabilità (a posteriori) che una sequenza di parole abbia generato la forma d'onda acquisita[8]. Questo approccio fu usato per la realizzazione di *IBM Tangora*, un sistema di riconoscimento vocale che, una volta allenato, era in grado di riconoscere circa 20 000 parole. Tuttavia le risorse computazionali necessarie erano troppo elevate per l'epoca, rendendolo di fatto un prodotto impossibile da commercializzare [31].
Successivamente l'introduzione di nuovi modelli linguistici come l'*n-gram* con *backoff* e ancora di più l'applicazione delle reti neurali sia deep che ricorsive unito alla realizzazione di hardware sempre più prestante ed efficiente portarono alla nascita e allo sviluppo sempre più rapido di soluzioni software commercializzabili, quali Google Voice e Siri. Inoltre, sempre grazie al deep learning, si stanno facendo passi avanti anche nel *Natural Language Understanding*. Ciò ha permesso la nascita delle *Conversational User Interfaces (CUI)*, interfacce con le quali l'utente interagisce in un linguaggio naturale, come se avesse di fronte un'altra persona [13]. Rispetto ad una GUI o ad un semplice sistema di riconoscimento vocale, le *CUI* offrono una curva di apprendimento praticamente nulla, poiché l'utente può esprimere il suo bisogno al sistema senza dover imparare comandi particolari. Esistono fondamentalmente due tipi di interfacce conversazionali, che si distinguono per la modalità di interazione: *chatbot* e *assistenti vocali*. Con questi ultimi l'interazione avviene prevalentemente via voce, mentre con i *chatbot* l'utente interagisce solitamente tramite una chat testuale. Se vogliamo semplificare, un assistente vocale può essere visto come un *chatbot* al quale è collegato un motore di *speech recognition* per l'input e uno di *Text-To-Speech* per l'output, come rappresentato nello schema in figura 2.6.

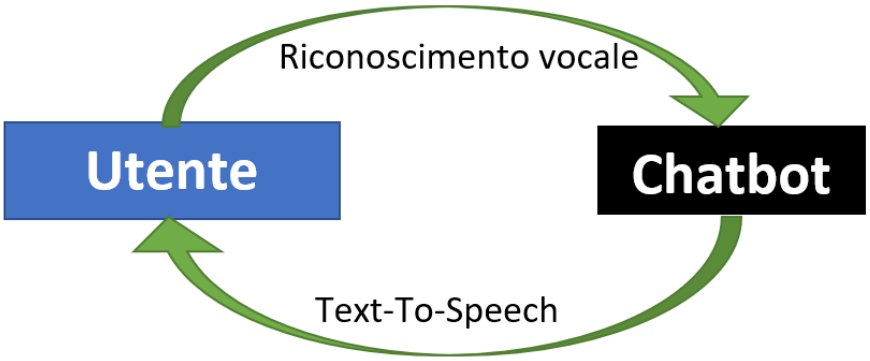


Figura 2.6: Visione del funzionamento di un assistente vocale rapportato ad un *chatbot*

Capitolo 3

Chatbot: caratteristiche e problematiche

Per potersi addentrare nel "mondo dei *chatbot*" è necessario prima comprendere alcuni termini fondamentali che permettono poi di descrivere in maniera più precisa le caratteristiche di un *chatbot* e delle sue modalità di addestramento.

Utterance Una *utterance* è un'espressione che l'utente usa o può usare per interagire con il *chatbot*. In altre parole una utterance è una frase che l'utente invia al *chatbot*: un esempio sono le frasi "Che tempo fa domani a Padova?" o "Quali sono le previsioni per domani a Padova?".

Intento Un intento è un'astrazione di una precisa intenzione dell'utente. Ad esempio le due *utterance* menzionate in precedenza sono tutte riconducibili ad un unico intento, cioè la richiesta di previsioni meteo per domani a Padova. Un intento ha le seguenti caratteristiche:

- **Trigger**, cioè le *utterance* con le quali l'utente esprime la sua intenzione;
- **Parametri**, ovvero le variabili di input. Ad esempio, nell'intento di richiesta di previsioni meteo, una variabile di input potrebbe essere la località per la quale l'utente vuole sapere le previsioni;
- L'**azione** o le azioni che il bot deve svolgere per soddisfare la richiesta dell'utente. Ad esempio, nel caso delle previsioni meteo, l'azione potrebbe essere quella di interrogare delle API di *OpenWeatherMap* per ottenere le informazioni desiderate;
- **Output**, cioè la risposta da restituire all'utente.

Entità Un'entità identifica il dominio di un parametro presente in un intento. Il concetto è simile a quello di *classe* nel mondo dell' *Object-Oriented Programming*: l'entità è la classe e il parametro è l'oggetto istanza di quella classe. Ad esempio, nell'intento di richiesta previsioni meteo, un ipotetico parametro *data* (indicante di quando si vogliono ricevere le previsioni) sarà istanza di un'entità *Data*;

Flusso conversazionale Il flusso conversazionale indica la logica della conversazione che il *chatbot* è in grado di gestire. In pratica indica il ragionamento che il bot è in grado di seguire. A livello grafico viene spesso rappresentato con un *decision tree* (un esempio viene fornito

in figura 3.1), anche se questa soluzione non è sempre l'opzione migliore. Un *decision tree* è una struttura rigida, soprattutto man mano che si scende di livello addentrandosi tra i vari percorsi, che male si adatta alla flessibilità di una conversazione. Inoltre, all'aumentare della complessità del dialogo e delle opzioni di risposta possibili ad ogni step, il numero di nodi e quindi di percorsi nell'albero di decisione aumenta esponenzialmente, rendendo complicata (se non addirittura irrealizzabile) la rappresentazione.[18]

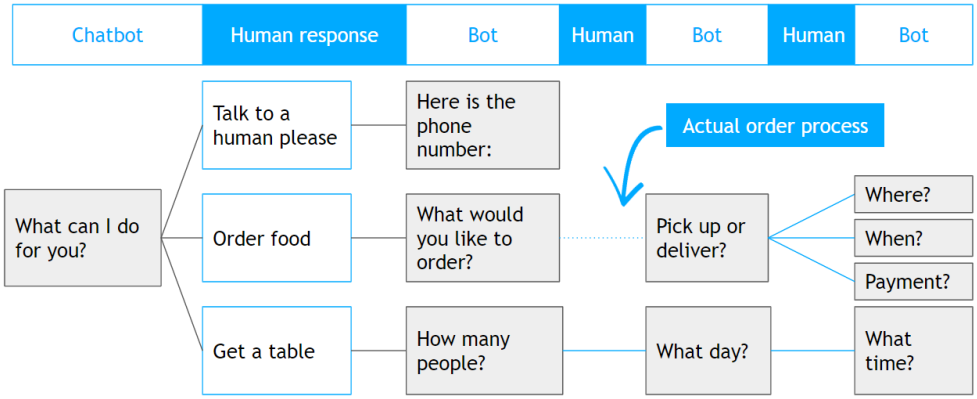


Figura 3.1: Esempio di decision tree come rappresentazione di un *chatbot*. Fonte https://bit.ly/3b1FXgf

3.1 Classificazione

E' possibile classificare un *chatbot* in base a numerosi criteri, anche se in letteratura se ne usano principalmente due: lo **scopo** e la **modalità di addestramento**

3.1.1 Scopo

In base allo scopo, possiamo distinguere due categorie: *chat-oriented* e *task-oriented* [38].

Task-oriented I bot di questo tipo sono realizzati per soddisfare un'esigenza specifica dell'utente, come prenotare un biglietto aereo od ottenere informazioni inerenti ad un particolare prodotto o servizio (FAQ). Essendo altamente specializzati, il dominio dei dialoghi che riescono a gestire è limitato. Una metrica utilizzabile per valutarne la bontà è il numero di turni conversazionali impiegati per soddisfare il bisogno: minore sarà questo valore, maggiore sarà l'efficacia del *chatbot*.

Chat-oriented I *chatbot* di questa categoria hanno come scopo riuscire a sostenere una conversazione complessa con l'utente. A differenza dei *task-oriented*, non ci sono vincoli sulla complessità, sulla struttura o sulle tematiche della conversazione. Molto spesso non operano secondo regole prefissate, ma imparano a rispondere applicando tecniche di NLU e Machine Learning ai dialoghi che li vengono sottomessi. Questo automatismo però può condurre a comportamenti inaspettati o indesiderati come dimostra il caso di *Tay*, un *chatbot* per *Twitter* realizzato da Microsoft basato su Machine Learning che in meno di una giornata è stato reso razzista. [9] Una delle modalità di apprendimento di questo *chatbot* era chiamata "repeat after me" e consentiva a *Tay* di apprendere le frasi che venivano usate dagli utenti per rispondere

ai suoi tweet, per poi riutilizzarle. Sfruttando questo sistema una larga comunità di utenti ha iniziato a creare dei tweet mirati di stampo fortemente razzista, antisemita o politicamente scorretti che *Tay* ha appreso ed ha iniziato a riutilizzare come risposte.

3.1.2 Modalità di addestramento

Una distinzione parallela alla precedente può essere fatta se al posto dello scopo viene considerato il criterio di addestramento, ovvero il modo in cui il *chatbot* "ragiona". A tal proposito, si possono distinguere due categorie: **rule-based** e **data-driven**.

Data-Driven I *chatbot* di questo tipo apprendono le risposte ai dialoghi grazie ad algoritmi di machine learning applicati a corpus e/o a conversazioni pregresse, sia tra due persone che tra persona e *chatbot*. Ad esempio *Tay* apprendeva da conversazioni che avvenivano su Twitter. Un altro esempio molto famoso di questo tipo è *Cleverbot* [23], un *chatbot* creato nel 2006 ma la cui intelligenza è stata addestrata a partire dal 1988. Sebbene siano molto potenti e con un grande potenziale, i *chatbot* di questo tipo soffrono di un problema, che li rende poco indicati ad essere utilizzati su ambiti sensibili: non si può fornire a priori una certezza sulla loro risposta. Ad esempio, se applicati in ambito sanitario, potrebbero dare un consiglio sbagliato sull'assunzione o sull'efficacia di un farmaco.

Hand Coded/Rule-Based Questi *chatbot* sono più limitati poiché operano in base a regole ben fissate inserite manualmente da un esperto. Siccome solo queste vengono utilizzate per gestire la conversazione, è possibile avere la certezza di come si comporterà il *chatbot* per ogni possibile espressione sottomessa dall'utente. Ecco che sono più indicati per gestire ambienti "sensibili", come appunto quello medico.

A dire il vero esiste anche una terza modalità di addestramento "ibrida" chiamata **crowdsourcing**. Il *chatbot* viene allenato inizialmente con un approccio *data-driven*. Successivamente le risposte che il *chatbot* allenato fornisce vengono analizzate da degli esseri umani, chiamati *coworker*, che hanno il compito di revisionare, correggere e completare i dialoghi, così da evitare risposte o comportamenti indesiderati [16].
L'idea di questo approccio parte da un dato di fatto: molto spesso i dati necessari ad addestare un *chatbot* su uno specifico task sono limitati. Ecco quindi che viene fatto ricorso a dei database pubblici, come le conversazioni su *Twitter* o i sottotitoli dei film, e le risposte generate dal *chatbot* vengono fatte analizzare a degli "esperti" umani, che provvederanno a correggere eventuali imprecisioni.

3.2 Il problema dell'addestramento

Come si può dedurre dalla sezione 3.1, la realizzazione di un *chatbot* è un problema molto complesso, che può essere scomposto in due sottoproblemi:

- *Realizzazione della piattaforma*, ovvero di quella componente in grado di interpretare la richiesta dell'utente. Essa può utilizzare tecniche di NLU per cercare di comprendere al meglio le richieste dell'utente e selezionare la risposta corretta da restituire ad esso. In altre parole, lo scopo principale della piattaforma è quello di mappare ogni *utterance* fornita in input nel corrispondente *bisogno*;

- *Addestramento*, ovvero insegnare al *chatbot* come rispondere ad ogni bisogno dell'utente.

Nell'ambiente della ricerca scientifica si stanno esaminando svariate soluzioni per rendere più efficace ed efficiente il processo di realizzazione di un nuovo *chatbot* (alcuni esempi di soluzioni "innovative" si trovano in [17] [14] e [12]), che tuttavia rimane un processo molto costoso in termini di risorse umane e temporali che richiede personale competente in ambito informatico e di analisi del linguaggio.
Recentemente però sul web hanno preso vita dei servizi il cui scopo è rendere il più semplice ed efficace possibile il processo di creazione e deploy di un nuovo *chatbot*. A tal proposito sono scesi in campo sia giganti dell'informatica quali *IBM*, *Amazon* e *Google*, che software house più piccole, come ad esempio *Intercom* e *ActiveChat*. Le soluzioni presenti sul mercato differiscono sia per le funzionalità che soprattutto per il target di utenti a cui sono rivolte. Si possono identificare due categorie di piattaforme: **generaliste** e **business-oriented**.

Generaliste A questa categoria appartengono piattaforme come *Google Dialogflow, IBM Watson e Amazon LEX* che permettono di realizzare bot molto complessi. I loro punti di forza sono sicuramente la potenza e l'accuratezza degli algoritmi di machine learning utilizzati per l'NLU , ma anche l'immenso numero di funzionalità che si possono integrare. Tuttavia il loro target è sicuramente quello di un informatico, o almeno una persona abituata al ragionamento di tipo logico. Molto spesso infatti è necessario utilizzare linguaggi di programmazione o tecnologie proprietarie, come AWS Lambda, per poter svolgere azioni anche all'apparenza molto semplici, come un'istruzione condizionale. Inoltre, dovendo offrire tante funzionalità, la loro interfaccia è più complessa da usare: puntano alla sostanza più che alla forma.

Business-oriented Lo scopo di queste piattaforme è offrire ad un'azienda uno strumento quanto più pronto possibile all'uso che permetta facilmente di gestire le casistiche più frequenti: "accoglienza" di un visitatore al loro sito, risposta alle FAQ e *lead qualification*. Nel primo caso il bot comparirà (generalmente come un'icona in basso a destra sullo schermo) in alcune o tutte le pagine del sito web aziendale con lo scopo di "accogliere" il visitatore, magari guidandolo nell'esplorazione del sito o nell'uso di funzionalità specifiche/avanzate (un esempio è riportato in figura 3.2). Le funzionalità che deve offrire sono basilari e molto spesso non accetta risposte testuali "aperte" ma solo a scelta multipla. A livello di addestramento, sono i bot più semplici da realizzare.

Riguardo i FAQ *chatbot*, una funzionalità molto importante che alcune piattaforme offrono è quella di importare in modo rapido una base di conoscenza esistente, in modo da permettere all'azienda di avere un bot (o almeno una prima versione) operativo in pochissimi minuti. Nel linguaggio dei *chatbot* intelligenti ogni FAQ rappresenta un intento e ogni risposta corrisponde alla sua azione di "fulfillment". In quest'ottica molte piattaforme (come Drift, Intercom o Chatfuel) permettono di applicare NLU in modo da rispondere a domande analoghe ma poste in maniera diversa rispetto a quelle presenti nella knowledge base. Questo comportamento, però, è facoltativo (e deve rimanere tale), poiché in alcune domande (penso all'ambito medico) non ci deve essere il rischio di fornire informazioni errate.
L'ultima tipologia di bot "mainstream" ha lo scopo di effettuare lead qualification, ovvero profilare il cliente/utente. Questo tipo di agenti deve essere in grado di categorizzare l'interlocutore in modo che possano poi essere a lui somministrati dei contenuti mirati. Un esempio è la suddivisione tra Sales Qualified Lead (cliente che acquista a colpo sicuro, che vuole esattamente quel prodotto, quindi poco sensibile alle campagne di marketing) e Marketing Qualified Lead (più propenso a diventare un cliente fidelizzato, quindi potenzialmente influenzabile con pubblicità/marketing mirato) [11]. La tipologia di bot in questione può essere sia rule-based, magari costruita fornendo all'utente poche e semplici domande (come la richiesta dell'indirizzo email) e/o interagendo con risposte a scelta multipla, sia "intelligente", quindi in grado di sostenere con l'utente conversazioni più complesse, meno delimitate e potenzialmente più lunghe.

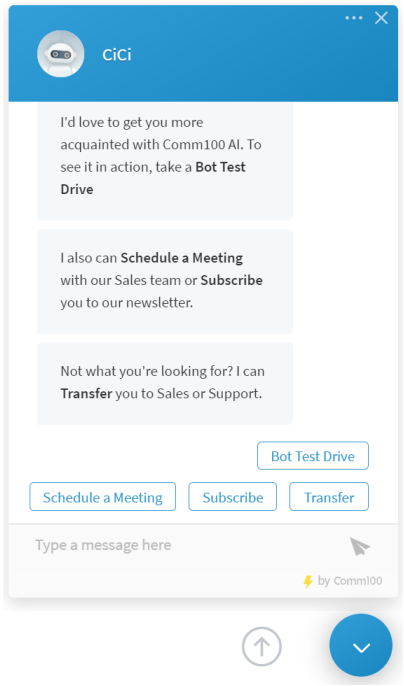


Figura 3.2: Esempio di un *chatbot business-oriented* per *lead qualification* tratto dal sito https://www.comm100.com

Capitolo 4

Stato dell'arte

Lo scopo di questo capitolo è presentare l'esito dell'attività di analisi di alcune delle piattaforme di addestramento *chatbot* attualmente presenti sul mercato. Questa attività ha avuto lo scopo di "toccare con mano" le soluzioni presenti sul mercato, in modo da carpirne pregi e difetti, traendone poi spunto per affinare le caratteristiche della piattaforma oggetto della tesi.
L'analisi ha toccato le seguenti piattaforme:

- *Business-oriented*: Intercom, DRIFT, Landbot, TARS, Chatfuel, Engati, ActiveChat, ManyChat e Microsoft QnA Maker;

- *Generaliste*: Amazon Lex, Google DialogFlow, IBM Watson.

4.1 Piattaforme Business-oriented

4.1.1 Intercom

Intercom [32] è un sistema di gestione della messaggistica aziendale, il cui scopo è raggruppare in un unico portale e ottimizzare al massimo le comunicazioni con i clienti. Secondo il sito web, le funzionalità offerte dalla piattaforma possono raggrupparsi in:

- *Supporto conversazionale*: grazie all'aggregatore di messaggi offerto dalla piattaforma, tutte le richieste di aiuto vengono raggruppate in un unico posto. Questo consente a operatori umani di rispondere in modo più efficiente e rapido all'utente in difficoltà. Inoltre ai messaggi in arrivo è possibile collegare un FAQ *chatbot*, che permette di rispondere in maniera automatica alle domande frequenti, scaricando gli operatori umani dai task più semplici;

- *Coinvolgimento conversazionale*: la piattaforma permette di analizzare il comportamento dell'utente durante la navigazione e in base a quello compiere delle azioni automatiche, come la comparsa di una chat collegata ad un *chatbot* o l'invio di email/notifiche personalizzate;

- *Marketing conversazionale*: è possibile costruire in maniera semplice dei *chatbot* per la lead qualification che possono tenere conto del profilo dell'utente e del suo rapporto passato con l'azienda.

4.1.1.1 Creazione di un *chatbot*

Intercom per quanto riguarda la creazione di un nuovo *chatbot* fornisce dei template da utilizzare come base di partenza: un elenco non esaustivo è fornito in figura 4.1.

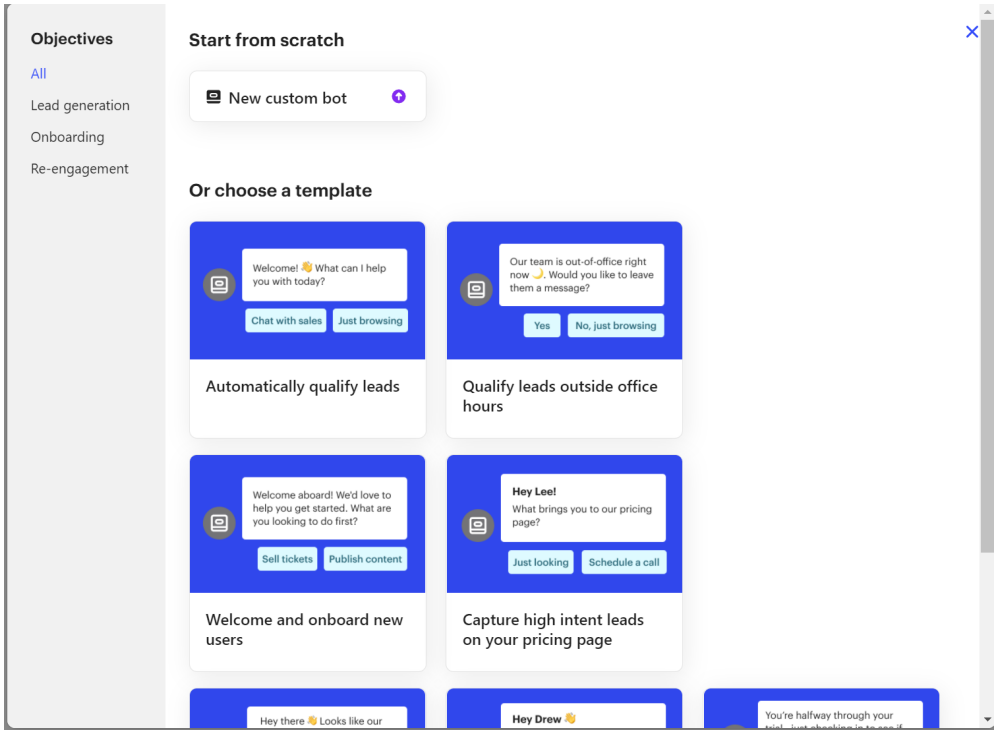


Figura 4.1: Schermata di selezione dei template in fase di realizzazione di un nuovo bot su Intercom

Per la creazione della logica del bot in maniera *code-free* (cioè per via grafica, senza usare un linguaggio di programmazione), si fa riferimento implicito al concetto di *intento*. Infatti le fattispece gestite dal bot sono raggruppate per scenario, come si vede dalla figura 4.2. Ogni scenario prevede più turni conversazionali e ad ogni *utterance* il bot può rispondere in maniera "semplice" (con del testo o delle risorse multimediali) oppure con una serie di bottoni.

Ad ogni bottone poi deve essere collegato un nuovo scenario (che corrisponde ad un nuovo intento), creando in questa maniera il *flusso conversazionale*. Infine si possono aggiungere delle "follow up actions", ovvero delle azioni da compiere al termine dell'intento. Queste possono essere semplici modifiche di attributi relativi all'utente o anche condizioni logiche più complesse (sempre definite in maniera visuale), come mostrato in figura 4.3. Queste condizioni logiche possono sfruttare anche i dati carpiti dal bot durante la conversazione tramite il settaggio di variabili di conversazione (chiamati "data attributes").

Da notare che i *chatbot* supportano solo risposte "secche" dell'utente (quindi o inserimento di testo o pressione di un bottone), ma non c'è nessuna possibilità di estrarre queste informazioni dall'*utterance* tramite tecniche di NLU : i *chatbot* creati non sono "intelligenti".

4.1.2 DRIFT

DRIFT [25] si definisce come una "piattaforma di accelerazione dei ricavi" ("Revenue Acceleration Platform"). Analogamente ad Intercom fornisce un servizio di aggregazione della

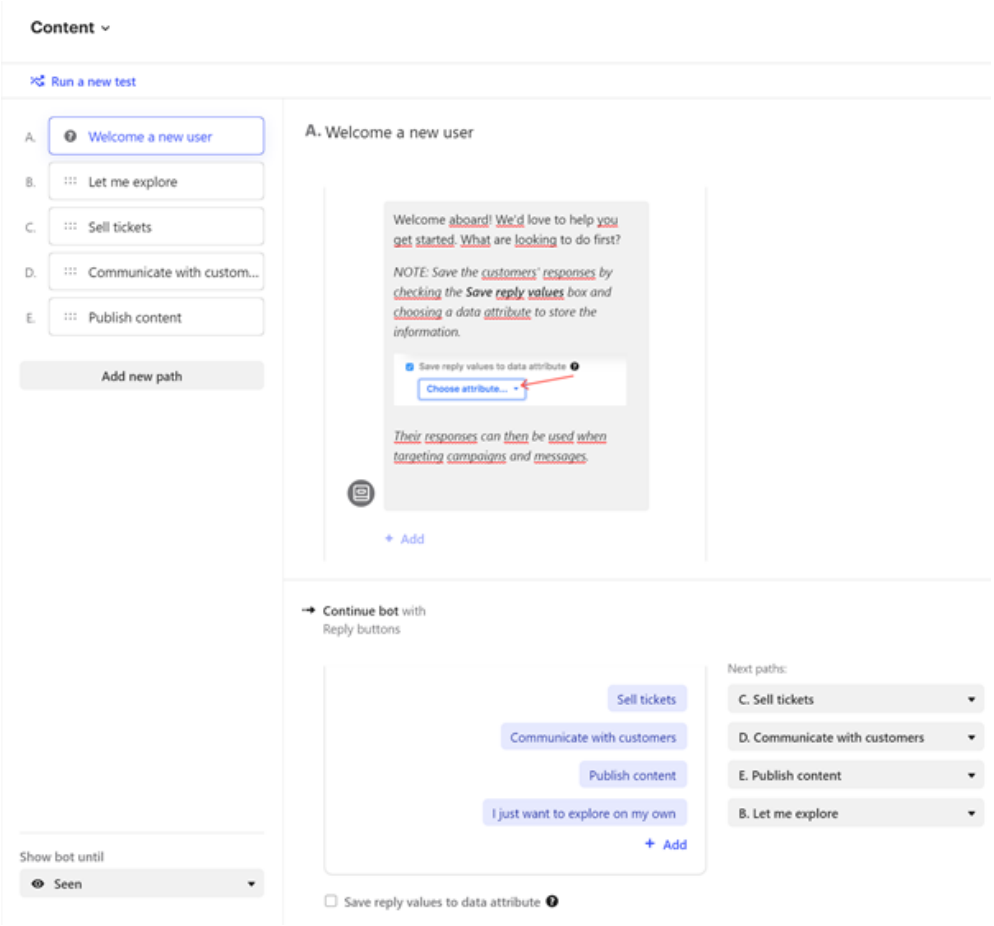


Figura 4.2: Creazione/modifica di un intento su Intercom

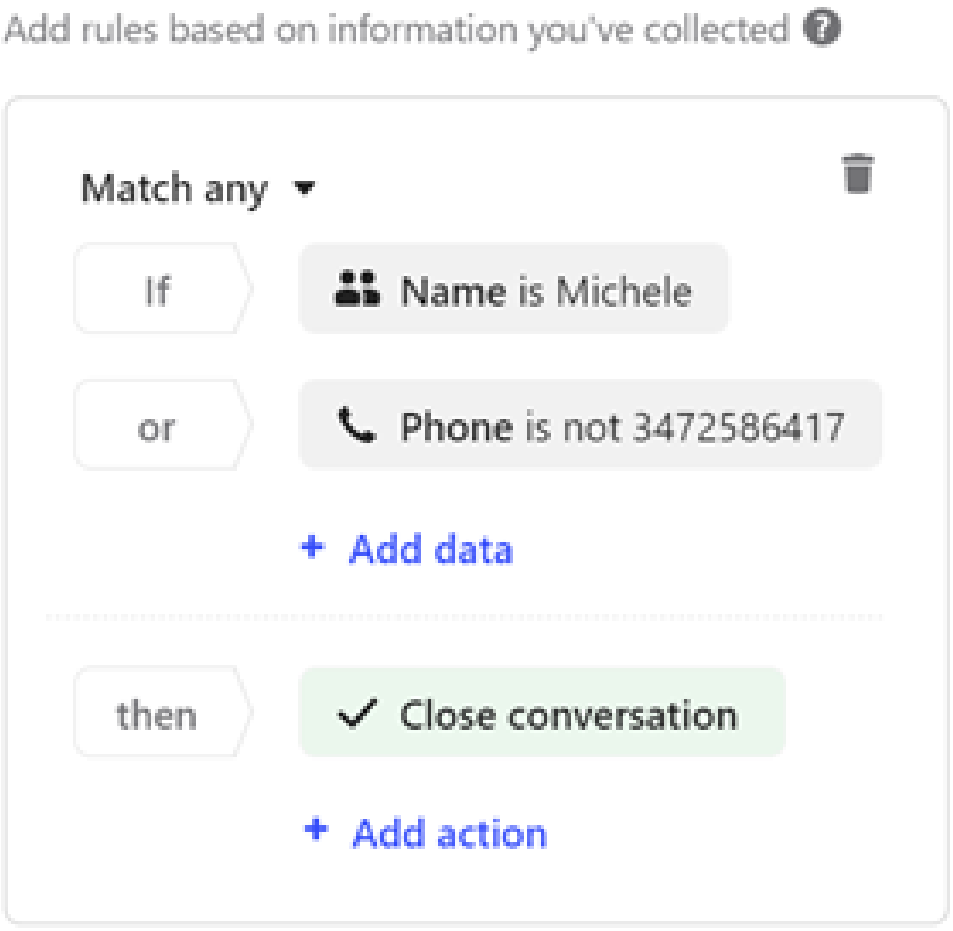


Figura 4.3: Follow-up action su Intercom

messaggistica, la cui gestione è possibile ottimizzare tramite soluzioni *chatbot*. DRIFT offre anche un servizio di creazione di *chatbot* intelligenti (chiamato DRIFT Automation), ma attualmente solo su commissione, cioè creati a mano dai loro esperti. La creazione guidata di *chatbot* quindi coinvolge solo quelli *rule-based* senza NLU e avviene utilizzando la metafora dei "mattoncini". Il flusso conversazionale viene rappresentato graficamente tramite un flow-chart (si veda figura 4.4) in cui ogni suo elemento ("mattoncino") rappresenta un'azione da compiere, come l'invio di un messaggio all'utente, la chiamata ad un'API esterna o l'aggiornamento di una variabile di sessione (ad esempio il salvataggio in una variabile del nome del cliente).

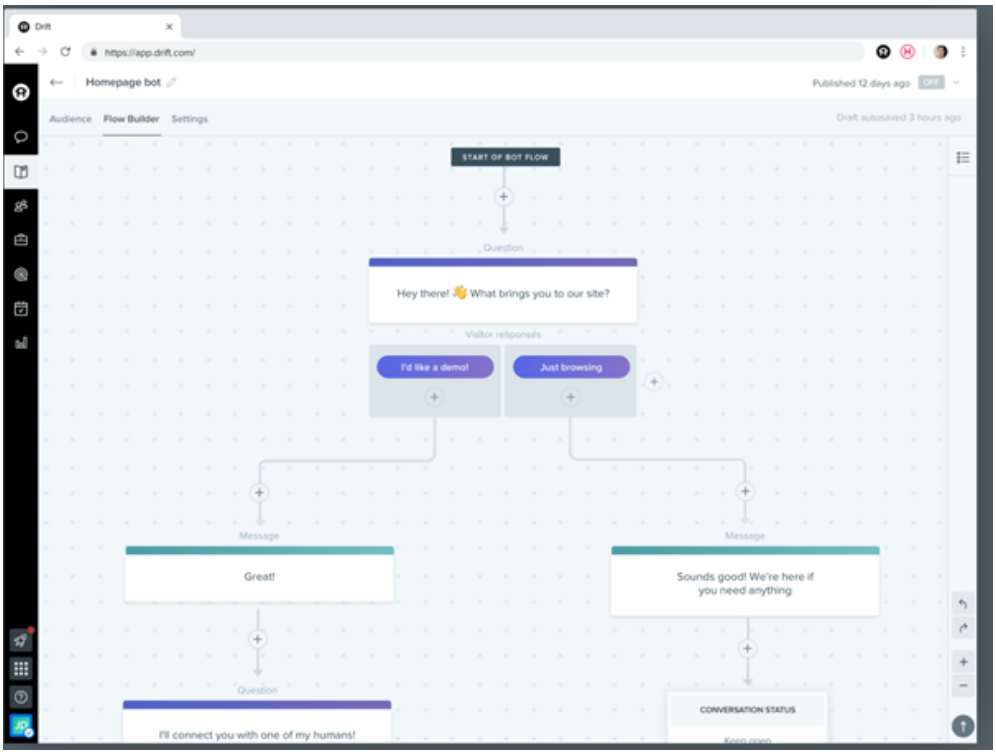


Figura 4.4: Schermata di addestramento di un bot in DRIFT

In tutto questo la differenza rispetto a Intercom sta proprio nella modalità di rappresentazione ed interazione con il flusso conversazionale, qua rappresentato sottoforma di flow chart.

4.1.3 Landbot

Landbot [33] è una piattaforma che permette di creare un *chatbot* disegnando in maniera guidata il flusso delle interazioni. Supporta integrazioni con servizi di terze parti come Slack, MailChimp, Google Sheets, Dialogflow.

In figura 4.5 è rappresentato lo schema dell'interfaccia di addestramento del bot: per la costruzione della logica la piattaforma si affida ai concetti di *block* e *arrow*, che riprendono la metafora dei "mattoncini". Ci sono molte tipologie di *block* già pronti che rispondono alle esigenze più comuni, come si può scorgere nella figura 4.6.

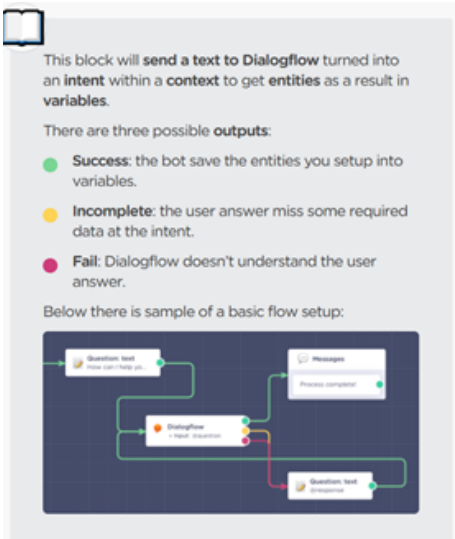


Figura 4.5: Schema dell'interfaccia di addestramento di un bot in Landbot

Inoltre sono presenti dei minivideo (della durata di un paio di minuti ciascuno) su youtube che spiegano il funzionamento degli elementi principali della piattaforma.

Ogni blocco presenta uno o più pallini in uscita, a ciascuno dei quali connettere una freccia per seguire un cammino diverso in base alla condizione che quel blocco rappresenta. Ad esempio, un blocco di tipo "Yes/No" ha due pallini, uno a cui connettere il percorso da seguire in caso l'utente scelga "Yes" e l'altro in caso l'utente scelga "No".

Infine (funzionalità ancora in beta) è possibile raggruppare i block in bricks in modo da creare l'equivalente di funzioni, che permettono di modularizzare la logica del bot (e anche tenere il flow-chart molto più ordinato).

In ogni caso sembra non sia presente la funzionalità di addestramento di un *chatbot* intelligente, se non utilizzando appunto l'integrazione con Dialogflow. Quest'ultima però semplicemente invia un input ad un bot esistente in Dialogflow e ne riceve l'output, senza fornire una modalità per addestrarlo.

4.1.4 TARS

TARS [41] permette di creare *chatbot* che funzionano tramite widget HTML oppure WhatsApp. Analogamente alle altre piattaforme analizzate fino ad ora, TARS presenta numerosi template da poter usare come base di partenza. Il flusso conversazionale viene gestito sotto forma di diagramma utilizzando la metafora dei "mattoncini", chiamati *Gambit*. Un *gambit* rappresenta un turno conversazionale completo, ovvero un messaggio posto dal *chatbot* all'utente e la relativa risposta (se richiesta).

Ogni gambit permette di raccogliere diverse tipologie di dato, quali (ad esempio) una risposta aperta, una a scelta multipla, file, data e ora, posizione geografica e altro, come mostrato nella figura 4.8. Inoltre un gambit può effettuare chiamate ad API esterne, integrarsi con Google Calendar e Zendesk chat.

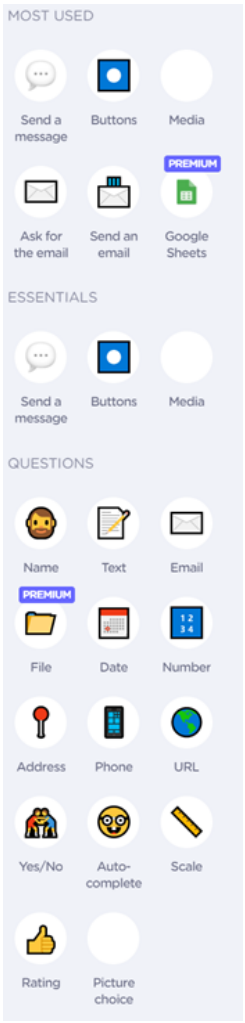


Figura 4.6: Tipologie di *blocks* disponibili su Landbot

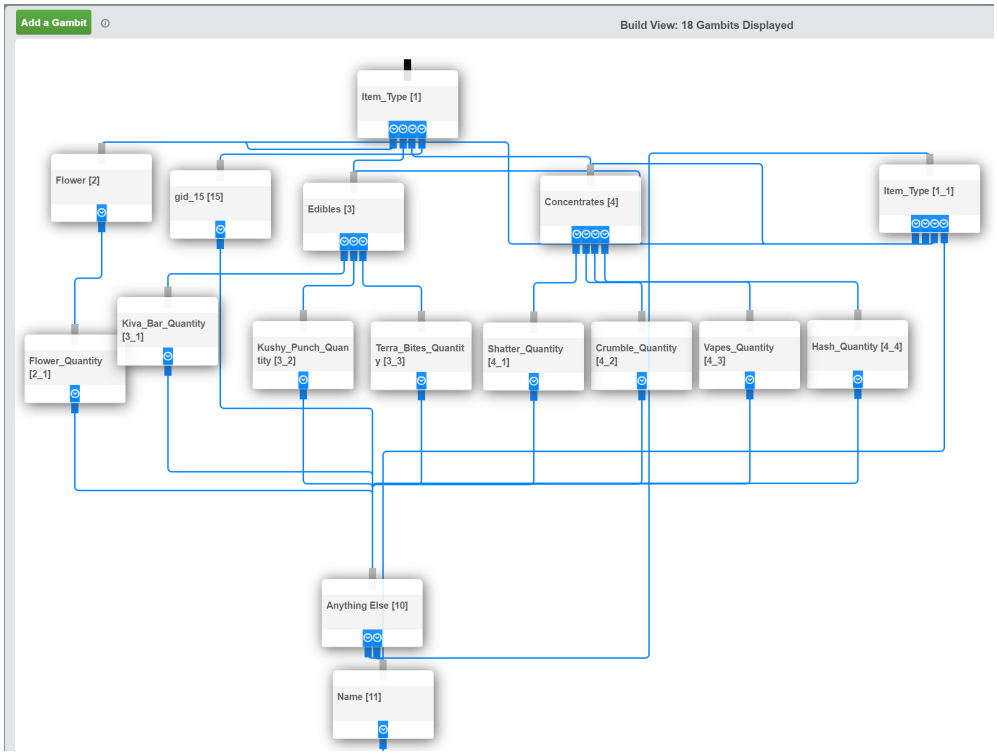


Figura 4.7: Flusso conversazionale su TARS

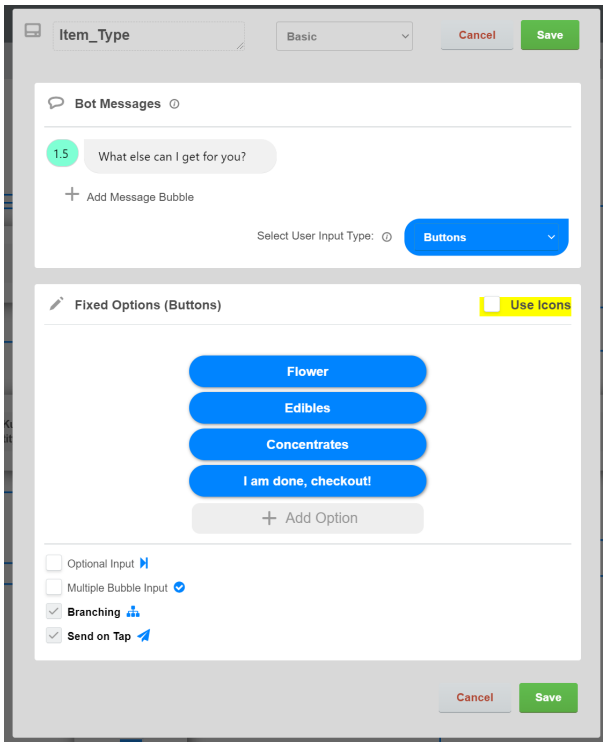


Figura 4.8: Dettaglio di un Gambit su TARS

In base alla tipologia, ogni gambit presenta uno o più outlet che corrispondono a condizioni sulla base dell'interazione con l'utente. Ad esempio nel *gambit* "*Edibles [3]*" (visibile nel secondo livello della figura 4.7) ci sono 3 outlet poiché si chiede all'utente di scegliere tra tre opzioni di risposta.
Rispetto le altre soluzioni presentate fino ad ora, l'approccio grafico di TARS presenta una criticità: i *Gambit* sono identificati con delle sigle. Questo aspetto rende sicuramente meno intuitivo l'utilizzo della piattaforma, soprattutto se un utente deve modificare un *chatbot* già esistente, in quanto dovrà imparasi il significato di ogni sigla.

4.1.5 ChatFuel

ChatFuel [22] ha come scopo facilitare la creazione di bot per Facebook, Instagram e Messenger. La tipologia di *chatbot* che permette di creare è *rule-based*, la cui logica viene gestita tramite flow-chart. Ancora una volta si fa rifermento alla metafora dei "mattoncini" per rappresentare i turni conversazionali, le azioni che il *chatbot* può compiere o le istruzioni condizionali che possono essere inserite nella logica del ragionamento. I mattoncini si possono collegare tra loro con un semplice drag-and-drop: ecco che si crea il flusso conversazionale, rappresentato in figura 4.9.

Le risposte che la piattaforma riesce a gestire sono di tipo testuale o a scelta multipla: non sono permesse invece risposte aperte, poiché è assente ogni forma di NLU .
Analogamente alle altre piattaforme, sono presenti dei template (alcuni anche a pagamento) da utilizzare per iniziare lo sviluppo. Un'istantanea della maschera di selezione del template è riportata in figura 4.10.

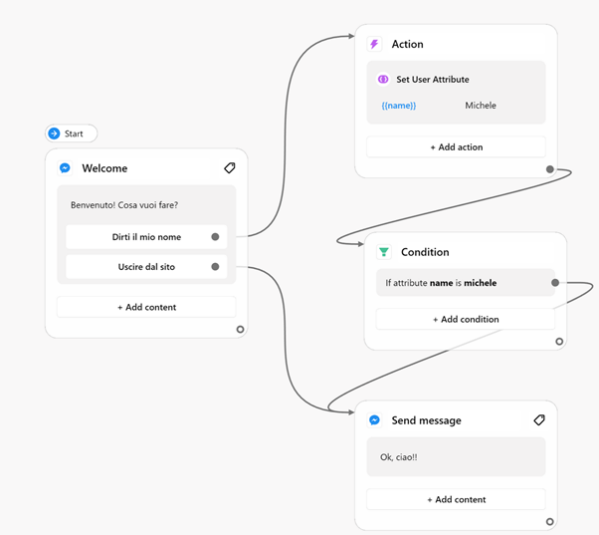


Figura 4.9: Flusso di conversazione in ChatFuel

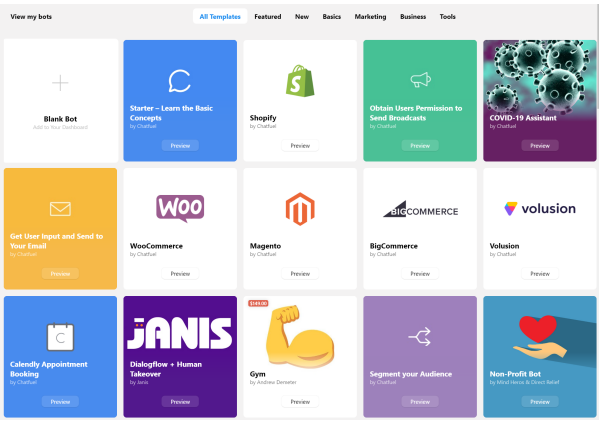


Figura 4.10: Selezione del template in ChatFuel

4.1.6 Microsoft QnA Maker

Microsoft QnA Maker [35] si propone di creare facilmente FAQ *chatbot*. La potenzialità di questa piattaforma è la facilità con cui è possibile creare la base di conoscenza del *chatbot*. Infatti, grazie all'IA permette di estrapolare dati direttamente da pagine web, file office, PDF e CSV.
In questa piattaforma si fa uso del concetto di *Knowledge base*, ovvero l'insieme delle conoscenze del *chatbot*. In questo caso, è la lista di tutte le FAQ a cui il bot sa rispondere. Ogni FAQ può essere identificata da più *utterance* e ammette una risposta. QnA Maker inoltre supporta le conversazioni su più turni, ovvero "domande che devono essere filtrate o perfezionate per determinare la risposta corretta" [24].
L'interfaccia è di tipo tabellare (figura 4.11). Ogni riga della tabella rappresenta una FAQ, nella parte sinistra sono presenti le frasi "trigger" (ovvero le *utterance* che il bot assocerà a quella domanda) mentre nella colonna di destra la risposta che verrà data dal bot.
Rispetto ai competitor visti fino ad ora, però, QnA Maker è più complicato da installare. Infatti richiede di configurare un account Azure creando un *service* apposito: non è sicuramente un'operazione che si può fare con un click.

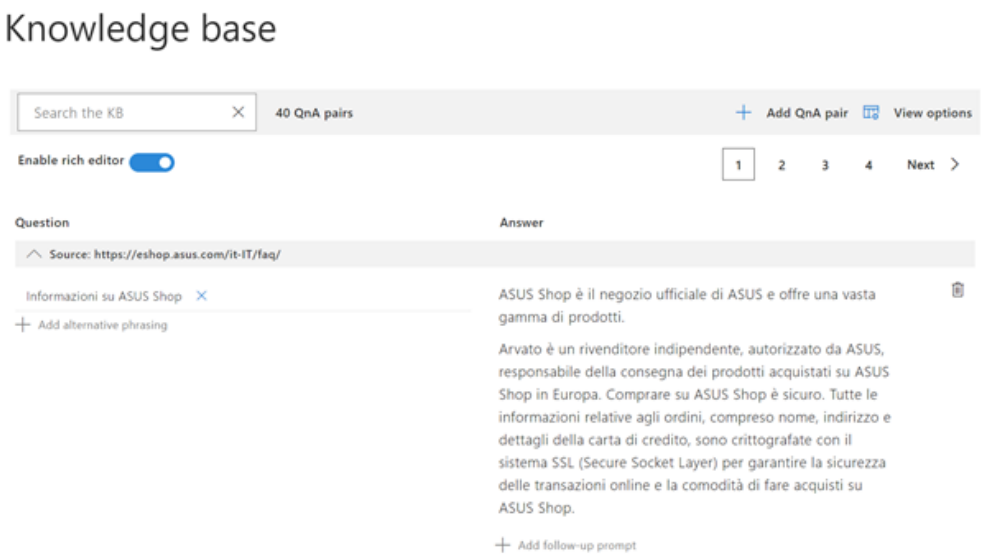


Figura 4.11: Base di conoscenza in Microsoft QnA Maker

4.1.7 Engati

Engati [27] è una piattaforma che presenta due caratteristiche peculiari: permette di creare *chatbot* che utilizzano NLU e prevede scorciatoie per l'addestramento di FAQ *chatbot*. Infatti, in modo simile a Microsoft QnA, è possibile addestrare il bot direttamente a partire da FAQ, senza dover configurare manualmente *utterance*, *intenti*, *entità* e il *flusso conversazionale*. A tal proposito, la piattaforma consente di importare una base di conoscenza già esistente a partire da una sorgente strutturata, cioè un file .xls, .xslx o .csv. Non consente di estrapolare informazioni invece da documenti o pagine web che non presentano una struttura definita a priori (cosa che è possibile fare in Microsoft QnA).

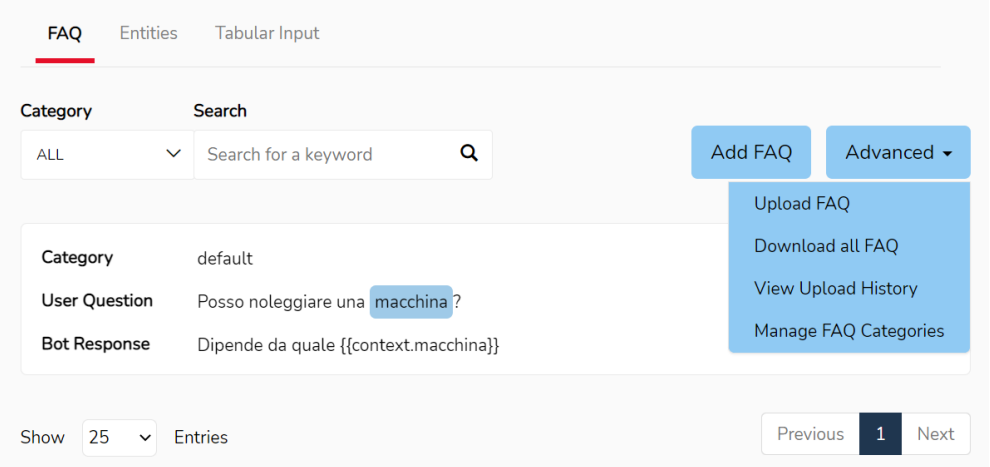


Figura 4.12: Gestione delle FAQ in Engati

Essa poi viene visualizzata su una schermata di riepilogo (figura 4.12) dove è possibile, tra le altre cose, categorizzare ciascuna FAQ in modo da semplificarne la gestione. Le FAQ inserite appaiono sotto forma di lista, Per ciascuna è visibile la categoria, la domanda e la risposta. All'interno della domanda ci possono essere parole evidenziate (nella figura 4.12 la parola "macchina"), che corrispondono alle entità.

Per inserire una nuova FAQ è disponibile la schermata in figura 4.13. In caso siano presenti entità nella domanda si devono evidenziare, e per ciascuna è necessario specificarne il tipo e l'obbligatorietà: se obbligatoria bisogna fornire anche il prompt utilizzato dal bot per richiedere l'informazione necessaria.

Quando si vada ad inserire o modificare un'entità, funzionalità offerta dalla maschera mostrata in figura 4.14, viene richiesto il nome, il tipo (dominio dei valori) e in base all'entity type selezionato, delle informazioni aggiuntive. Ad esempio, per un *entity type* di tipo *Date* (cioè una data) viene richiesto come si intenderà poi interagire con la data (se con un campo di testo oppure con un picker); per il tipo "Custom Values" viene richiesto di inserire i valori possibili che l'entità può assumere e per il tipo "Regex Pattern" viene chiesto di specificare quale sia l'espressione regolare da controllare.

4.1.8 ActiveChat

ActiveChat [19], in modo simile ad Engati, permette di creare *chatbot* che utilizzano NLU , il tutto tramite un'interfaccia grafica, quindi *code-free*. Per fare ciò la piattaforma richiede di inserire due tipologie di informazioni: gli *intenti* e le *skill*. Queste ultime indicano le azioni che il sistema dovrà svolgere una volta riconosciuto l'intento associato.

Nel dettaglio, come mostrato in figura 4.15, un intento è composto da una lista di *utterance* (cioè le frasi trigger) e da un'azione che verrà svolta una volta riconosciuto. L'azione può essere di due tipi: invio di una risposta testuale semplice (come in figura 4.15) oppure l'invocazione di una *skill*, che facendo un parallelismo con il mondo della programmazione corrisponde ad una funzione. Per la costruzione di una skill viene usata la metafora dei "mattoncini" (si veda figura 4.16), ovvero viene chiesto di costruire un diagramma di flusso formato da blocchi di diversa tipologia, tra i quali:

- *Event*, che riguardano la gestione degli eventi presenti in ActiveChat. Nello specifico,

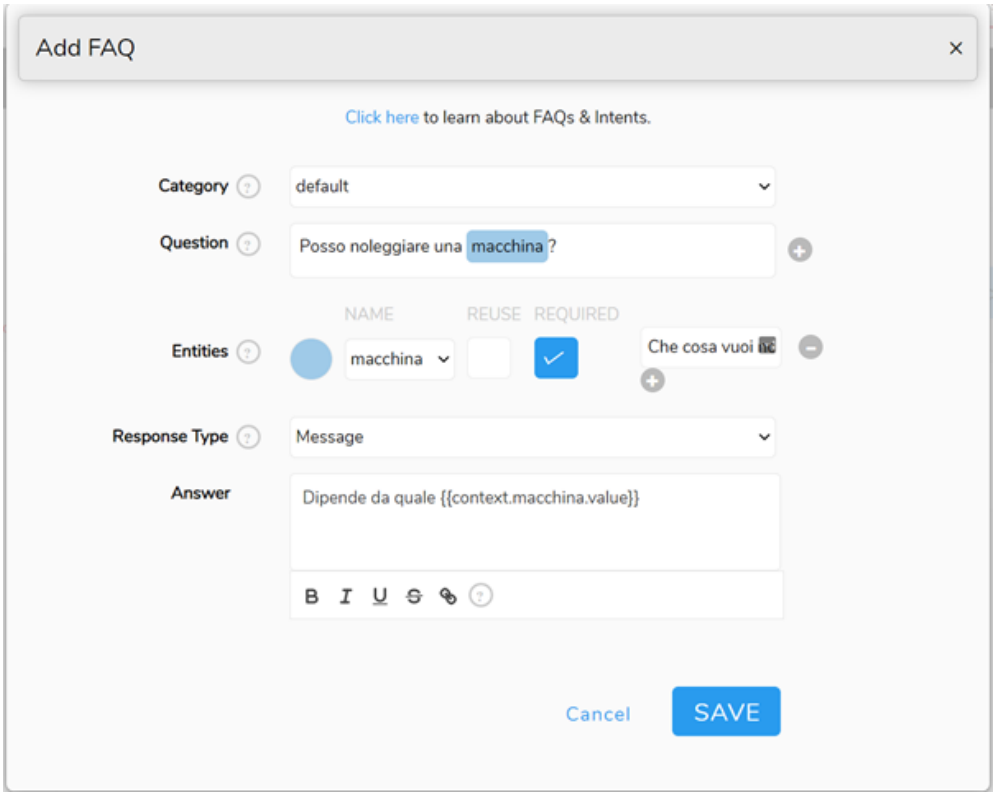


Figura 4.13: Inserimento di una nuova FAQ in Engati

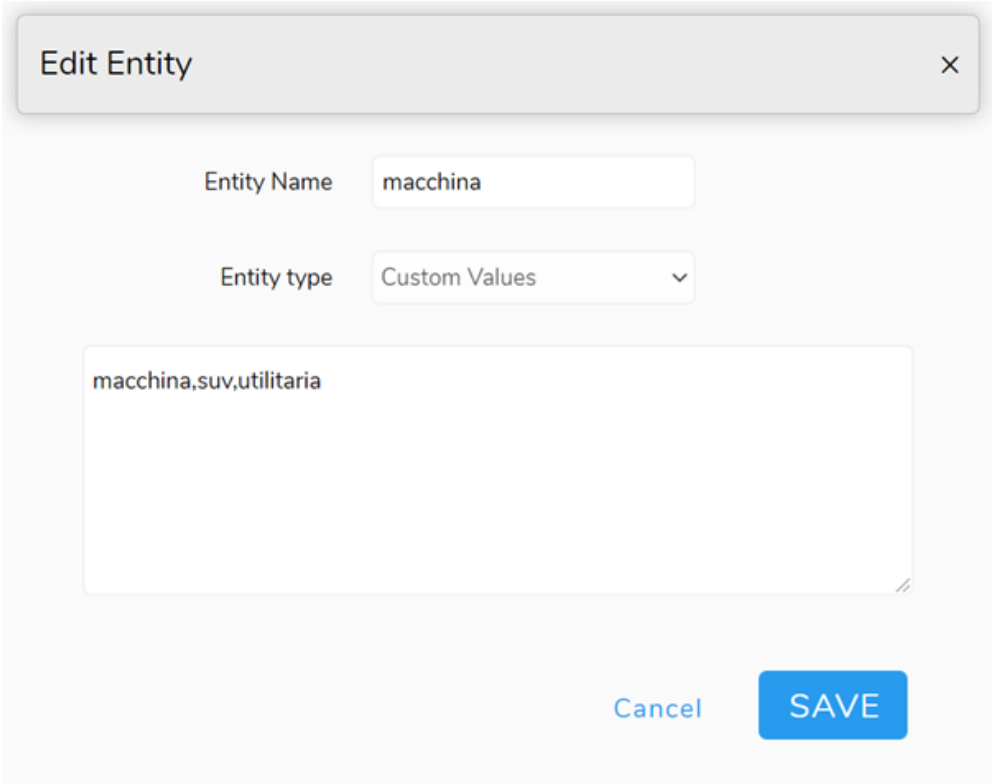


Figura 4.14: Modifica di un'entità in Engati

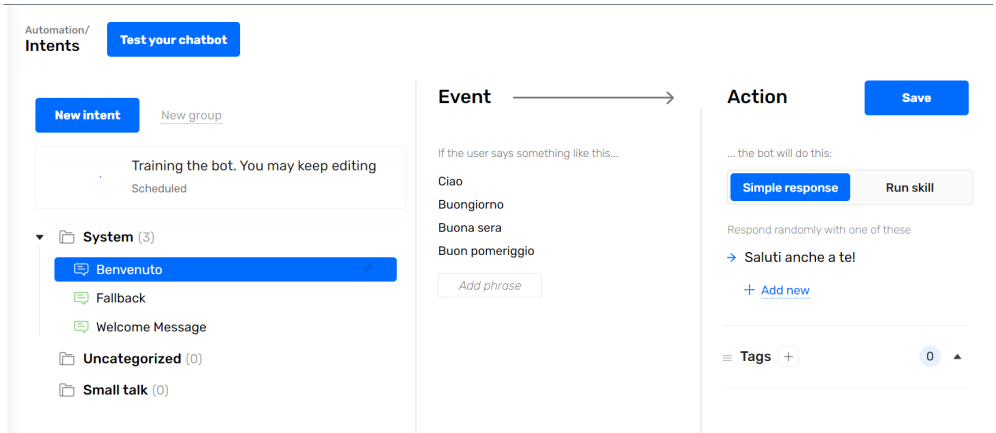


Figura 4.15: Modifica di un intento in ActiveChat

ogni skill inizia con un blocco *Catch*, che equivale ad un listener per quella skill. E' possibile passare da una skill all'altra tramite un blocco *Send*, generando un evento che verrà intercettato dal *Catch* di un'altra skill. Molto simile per funzionamento è il blocco *Trigger*, che permette di generare un evento intercettabile dal *Catch* di un'altra skill di un'altra istanza di chat: in pratica permette di mettere in comunicazione più chat, quindi più utenti. Infine, l'ultimo blocco è *Lead*: permette di inviare in modo rapido i dettagli dell'utente con cui il bot sta interagendo alla propria mailbox;

- *Logic*, che contiene il blocco *Switch* (istruzione if-then-else sul valore di una variabile), Data (che permette di assegnare/modificare valori delle variabili) e *Validation*, che permette di controllare se il valore di una variabile rispecchia una certa struttura (ad esempio se una stringa è un indirizzo email);
- *NLP*, il cui scopo è quello di analizzare un input testuale dell'utente affidandosi al Natural Language Processing Engine di *Dialogflow*.

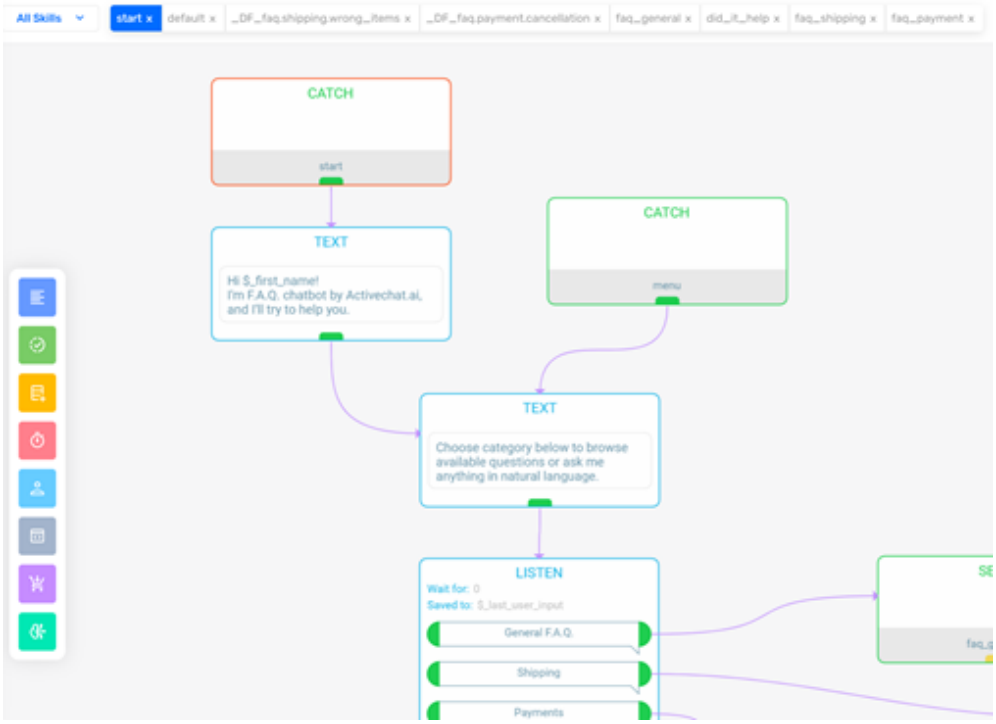


Figura 4.16: Modifica di una skill in ActiveChat

Nell'ottica della creazione di un FAQ *chatbot*, questo approccio risulta abbastanza macchinoso: per ogni FAQ è necessario definire un intento manualmente, dove le *utterance* indicano le frasi "trigger" di quella FAQ e la risposta viene inserita come "Simple Response" sotto la voce "Action". Inoltre, per utilizzare NLU ActiveChat si affida a *DialogFlow*, quindi non possiede un vero e proprio motore NLU integrato.

4.1.9 ManyChat

ManyChat [34] nasce come una piattaforma che permette alle aziende di raggruppare diversi flussi di conversazione in un unico posto (un po' alla Blackberry Hub). Attualmente supporta

connessioni con Facebook Messenger, SMS ed email.
Una delle funzionalità della piattaforma però è anche l'automazione: permette di interagire in modo (semi)automatizzato con l'interlocutore (attualmente solo via Facebook) tramite la definizione di uno o più bot.
La logica di gestione della conversazione da parte del bot è definita in maniera visuale tramite un diagramma di flusso con il paradigma dei "mattoncini".
Un flusso di dialogo è chiamato *Flow* ed è composto da diversi step (mattoncini), tra i quali c'è sempre uno "Starting Step" (punto di partenza). Un esempio di flusso è rappresentato in figura 4.17. Ogni mattoncino/blocco ha una tipologia, che ne definisce la funzionalità: invio di un contenuto (messaggio/immagine) all'utente, raccolta di una sua risposta, istruzione condizionale su i dati raccolti e lancio di azioni (come ad esempio l'invio di una mail/sms, la catalogazione di una conversazione, la chiusura del bot ecc…).

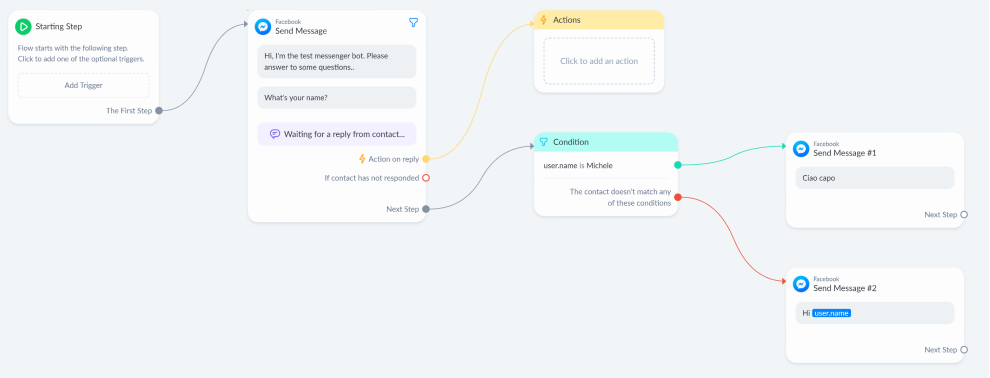


Figura 4.17: Flow Builder presente in ManyChat

A differenza di molti dei competitor, viene data la possibilità di costruire il bot anche tramite un "basic builder" (si veda figura 4.18), ovvero un'interfaccia con layout a lista.

Analogamente ad ActiveChat, anche ManyChat richiede di definire in che modo ogni flow verrà eseguito, cioè a che eventi esso risponderà. Tuttavia, a differenza del competitor, gli eventi disponibili sono molti meno: reazione ad una o più parole chiave presenti in un messaggio, data/ora specifica, variazione del valore di un parametro utente o di sistema.

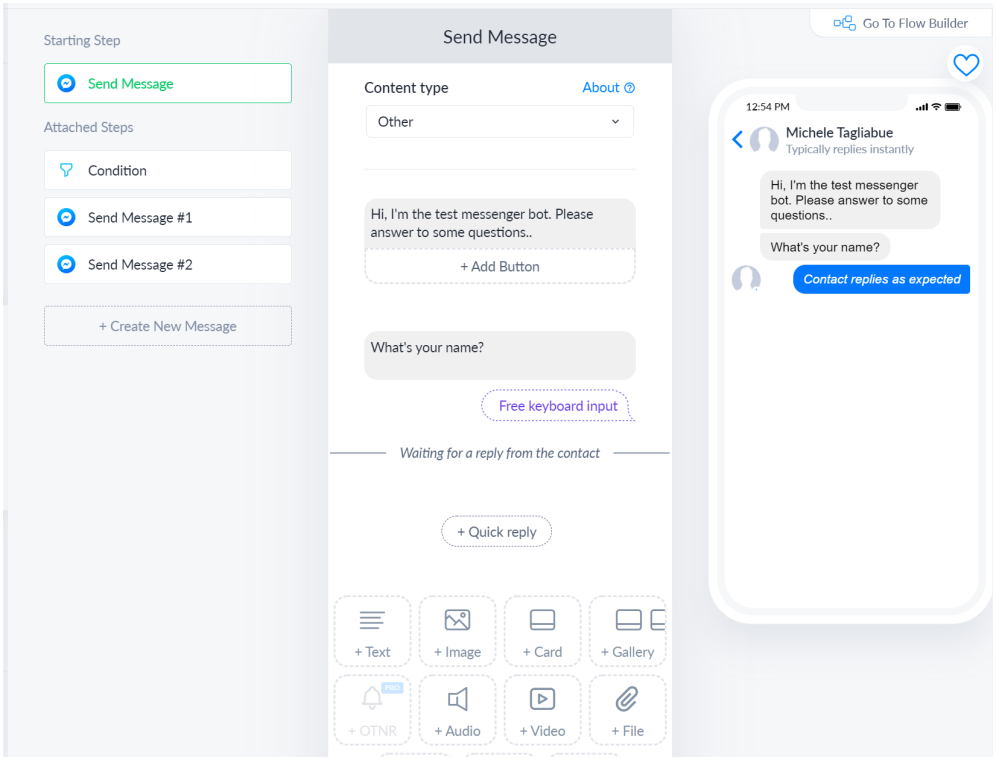


Figura 4.18: Basic Builder presente in ManyChat

4.2 Piattaforme generaliste

4.2.1 Amazon Lex

Amazon Lex [20] è un servizio facente parte dell'ecosistema AWS che permette di creare *chatbot* intelligenti sfruttando il motore NLP /NLU alla base di *Alexa*. Lex permette di integrare i suoi bot con varie piattaforme quali Slack, Facebook Messenger e Twilio (SMS) in modo facile tramite delle funzionalità ad-hoc presenti nel pannello di amministrazione.

In Amazon Lex un *chatbot* è formato da **Intents** (intenti) e **Slot Types** (entità). Una caratteristica rilevante è che queste due tipologie di elemento sono condivise in tutta la piattaforma, un intento può essere riutilizzato da più bot e un'entità da più intenti.

Entità In Lex le entità vengono chiamate *Slot Types* ma il concetto è sempre quello: sono dei tipi di dati personalizzati, molto simili a delle enumerazioni nel mondo dell'OOP. Ogni entità ha i seguenti attributi:

- *Nome*, che deve essere univoco in tutta la piattaforma. Il nome non può contenere caratteri speciali, spazi e numeri, per cui è più un codice che un nome descrittivo.
- *Value*, ovvero i valori che l'entità può assumere. Ad esempio un'entità chiamata "CarType" che identifica la tipologia di veicolo da poter noleggiare potrà avere tra i propri valori "SUV", "Utilitaria", "Auto Sportiva", "CrossOver".
- *Slot Resolution*, cioè la strategia da adottare per il riconoscimento dei valori delle entità. Il valore di ogni entità potrebbe essere espresso in vari modi dall'utente. Ad esempio una macchina di tipo "Utilitaria" potrebbe anche chiamarsi "city car", oppure un "SUV" "fuoristrada". Per far riconoscere questi valori a LEX ci sono due possibilità (strategie):
  - *Expand Values*: lasciare libertà a LEX di capire quando una parola è sinonimo del valore di un'entità, grazie al suo motore di NLU ;
  - *Restrict to Slot values and Synonyms*: i valori degli slot verranno riconosciuti solo se identici a quelli forniti. Se questa strategia viene selezionata, per ogni valore di slot è possibile (e opportuno) fornire anche una lista di sinonimi.

Intenti Ogni intento, la cui schermata di inserimento/modifica è visualizzata in figura 4.19, presenta le seguenti caratteristiche:

- *Nome*, che serve per identificarlo univocamente. Anche in questo caso, come negli *slot types*, il nome è più che altro un codice: è soggetto agli stessi vincoli sintattici e deve essere univoco in tutto l'account, a prescindere dal bot a cui l'intento è associato.
- *Slots*: indicano le entità necessarie all'intento. In altre parole, uno *slot* indica un'informazione che deve essere raccolta per poter portare a compimento l'intento. Facendo un paragone con il mondo dell'OOP, uno slot è una variabile di classe *Slot Type* Nell'esempio del noleggio di un'automobile, degli slot potrebbero essere il tipo di automobile che si intende noleggiare, l'età del conducente, il periodo (data di inizio - data di fine) del noleggio. Per ogni slot è necessario definire:
  - *Nome*, che deve essere univoco nel contesto dell'intento e non contenere spazi, numeri o caratteri speciali al di fuori di "_";

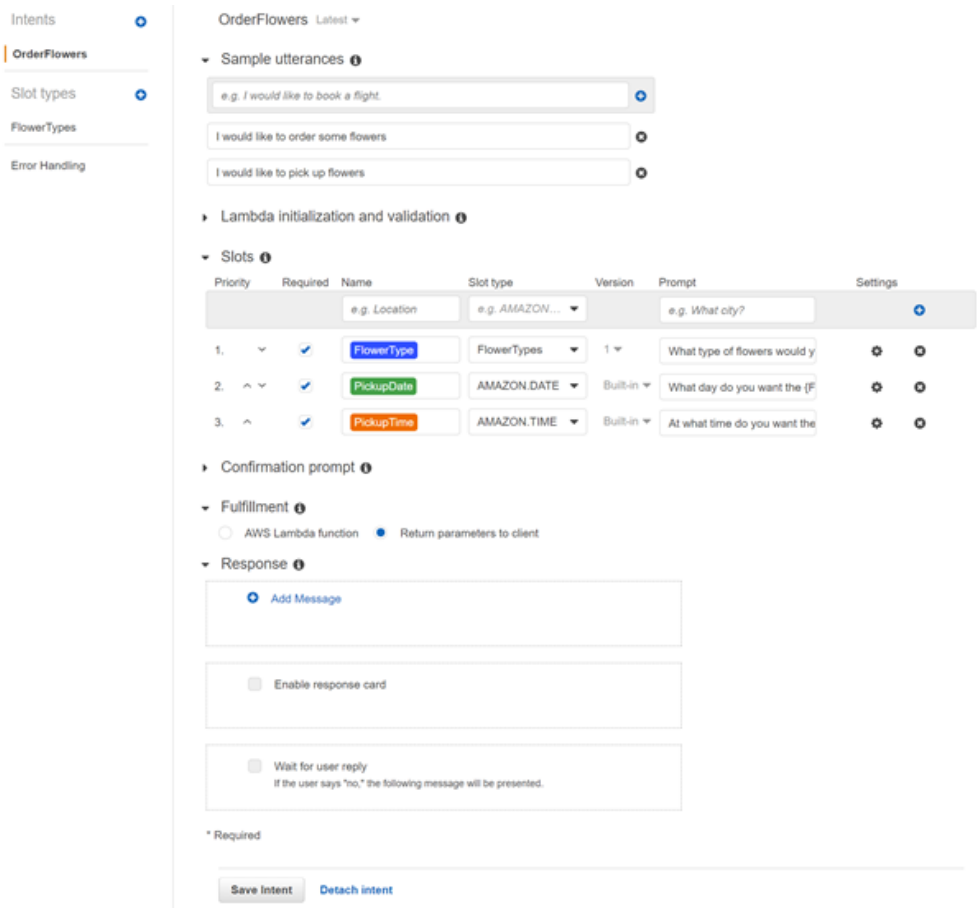


Figura 4.19: Schermata di modifica di un intento in Amazon Lex

  - *Slot type*, ovvero l'entità associata allo slot, la "classe" della variabile corrispondente allo slot;
  - *Prompt*, la frase che verrà utilizzata per richiedere all'utente l'informazione associata allo slot. Ad esempio, nel caso dello slot *CarType* (che contiene la tipologia di macchina da noleggiare), un possibile prompt potrebbe essere "Quale tipo di automobile vorresti noleggiare?";
  - *Priorità*, ovvero l'ordine con il quale verrà richiesta all'utente l'informazione corrispondente a quello slot, se non è stato possibile estrarre l'informazione da una *utterance* precedente;
  - *Required*, flag che indica se è obbligatoria la valorizzazione dello slot da parte dell'utente

- *Sample utterances*, cioé le frasi "trigger" che Lex utilizzerà per riconoscere l'intento. All'interno di ogni frase è possibile evidenziare gli slot per permettere a Lex di valorizzarli in automatico. Ritornando all'esempio del noleggio dell'automobile, alcune *utterance* potrebbero essere "Vorrei noleggiare una macchina" oppure "Vorrei noleggiare un'utilitaria". In quest'ultima frase è presente un'informazione in più: il tipo di macchina (corrispondente allo slot *CarType*) che si intende noleggiare. Evidenziando la parola "utilitaria" è possibile associarla allo slot *CarType*, in modo che Lex riesca a valorizzare in automatico tale slot quando l'utente esprime il suo intento con questa *utterance*;

- *Lambda Initialization and Validation*: funzione AWS Lambda che viene invocata ogni volta che l'utente interagisce con il *chatbot*. Questo consente di gestire delle fattispecie complesse, magari interrogando un database o effettuando delle chiamate ad API remote;

- *Fulfillment*, cioè l'azione che deve essere intrapresa al termine dell'intento, cioé quando è stata riconosciuta una *utterance* e tutti gli slot obbligatori sono stati valorizzati. Questa azione può essere di due tipi: chiamata ad una funzione AWS Lambda oppure restituzione di un messaggio;

- *Response*, la risposta che viene ritornata all'utente al compimento dell'intento, se questo è il comportamento impostato alla voce *Fulfillment*;

- *Confirmation prompt*, ovvero una frase (facoltativa) che verrà utilizzata per confermare l'intento. In pratica, quando tutti gli slot obbligatori sono stati valorizzati, il bot, prima di procedere all'invocazione dell'azione di *fulfillment*, può chiedere all'utente una conferma finale tramite la frase da inserire in questa sezione.

A livello di interfaccia si può notare dalla figura 4.19 che non è presente nessun tipo di "ausilio grafico" (come un diagramma) per la visualizzazione del flusso di conversazione. Inoltre ogni tipo di logica di business è demandato alle funzioni AWS Lambda associate agli intenti. Questo aspetto rende la creazione di un *chatbot* che effettui un minimo di "ragionamento" (anche una semplice istruzione condizionale sull'input ricevuto) decisamente fuori dalla portata di un soggetto senza esperienza nel mondo della programmazione.

Una funzionalità molto interessante offerta da Amazon Lex invece è l'importazione e l'esportazione di bot, intenti ed entità tramite file JSON. Tra le altre cose, questa funzionalità consente di creare un *chatbot* esternamente (magari con un'altra piattaforma) ed importarlo in modo semplice in Lex.

4.2.2 Google Dialogflow

Google Dialogflow [30] è la soluzione offerta da Google per la creazione di *chatbot* intelligenti. I bot creati con questa piattaforma possono sfruttare tutta la potenza del motore di NLU che sta alla base di tutti i servizi dell'azienda, tra i quali *Google Assistant* (l'assistente digitale di Google). Attualmente *Dialogflow* è offerto in due edizioni: *ES* e *CX*.

4.2.2.1 Dialogflow ES

ES, sigla che sta per "Essential", è la versione "legacy" di Dialogflow. Secondo la pagina ufficiale [30] questa versione è indicata per la realizzazione di *chatbot* "semplici". La struttura di un bot è sempre basata sul paradigma intento/entità usato, ad esempio, anche da Amazon Lex e IBM Watson. Nel dettaglio, un *chatbot* possiede i seguenti elementi caratteristici:

- *Intento*: intenzione dell'utente per un turno di conversazione. Un intento è definito da:
  - *Frasi di training*, che servono per addestrare il motore NLU affinché sia in grado di comprendere l'intenzione dell'utente;
  - *Azioni* che vengono scatenate quando l'intento viene riconosciuto (ad esempio aggiunta di un appuntamento al calendario, invio di un'email, chiamata ad un API esterna);
  - *Parametri* (entità) che vengono estrapolati dalla frase trigger dell'intento. Ad esempio, nella frase "Voglio noleggiare un SUV per domani", le parole SUV e domani rappresentano due parametri (il primo di tipo macchina e il secondo di tipo data). Piccola nota: a differenza di LEX, Dialogflow è in grado di estrarre i parametri direttamente dalla frase trigger dell'intento;
  - Insieme di *risposte* che verranno restituite all'utente finale. Possono essere messaggi di testo semplici, elementi multimediali o ulteriori domande (chiarimenti).
- *Entità*: rappresenta il tipo di ogni parametro, analogamente a quanto già visto con le altre piattaforme. Dialogflow fornisce entità di sistema e permette di definirne di personalizzate.
- *Contesto*: Indica il contesto della frase. Ad ogni intento è possibile associare un *output context*, ovvero il contesto che si attiverà nel momento in cui l'intento viene scatenato. Ad un intento poi è possibile associare un *input context*: quando un contesto è attivo e l'utente si rivolge al bot, Dialogflow aumenterà la probabilità di invocare gli intenti aventi come contesto attivo *input context* (per un'illustrazione del funzionamento si veda figura 4.20).

  La definizione del contesto corrente viene semplificata con la possibilità di usare i *follow-up intent*, ovvero di associare ad un intento "padre" uno figlio. Così facendo verrà creato un contesto e collegato come *output context* dell'intento padre e *input context* dell'intento figlio. Il contesto serve anche da memoria: referenziando il contesto è possibile accedere a tutti i valori dei parametri degli intenti già soddisfatti aventi lo stesso contesto. Questo è molto utile nel caso degli intenti *follow-up*: se si acquisisce un parametro nell'intento padre, sarà possibile accedervi dall'intento figlio con *#nomeContesto.nomeParametro*.

4.2.2.2 Dialogflow CX

CX, acronimo di *Customer Experience*, è stato concepito per la gestione di conversazioni molto complicate con l'utente, che possono seguire percorsi multipli e densamente connessi tra loro,

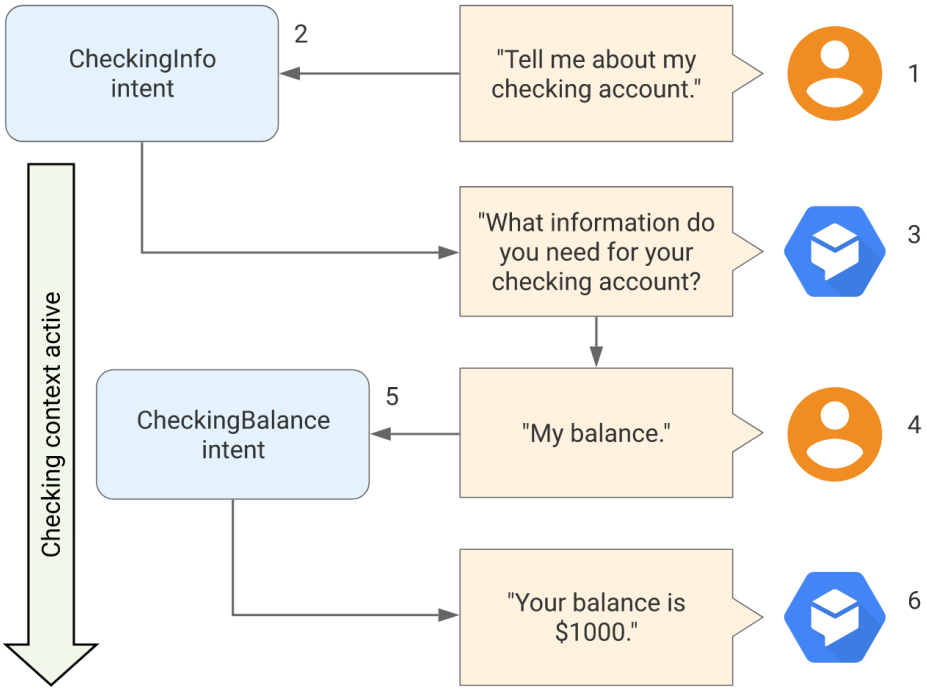


Figura 4.20: Funzionamento del contesto in Dialogflow ES. Fonte: https://bit.ly/3lqHlN4

quindi non rappresentabili direttamente tramite un semplice albero n-ario di dimensione finita. Per questo adotta un paradigma diverso da quello "intento/entità" visto nelle altre piattaforme, tra cui Lex (e IBM Watson), che si basa sui seguenti concetti:

- *Agent*: l'agente (bot) è visto come un automa a stati.
- *Flows (flussi)*: rappresentano degli argomenti della conversazione. Secondo la documentazione ufficiale, nell'esempio di un bot che ordina la pizza, i flussi che potrebbero essere presenti sono: informazione del cliente (ad esempio dire qual è il menù), raccolta dell'ordinativo e conferma dell'avvenuto ordine. Ciascuno di questi tre elementi rappresenta una funzionalità a sé stante, che può quindi essere modularizzata: un flusso. Secondo Google, il vantaggio è che così facendo si riesce a suddividere al meglio le responsabilità di sviluppo e manutenzione delle varie parti del bot tra più soggetti/team, specialmente quando la fattispecie da modellare risulta complessa.
- *Pages*: sono gli stati in cui si può trovare l'agente. In pratica servono a mantenere lo stato della conversazione, ruolo simile al contesto in Dialogflow ES, Watson e Lex. Un flusso al suo interno contiene diverse pages, in quanto un flusso rappresenta un'intera conversazione e gli stati rappresentano i "punti" in cui la conversazione si può trovare. Ad ogni istante, essendo l'agente un automa a stati finiti, solo una pagina (solo uno stato) può risultare attivo. In ogni page si possono raccogliere delle informazioni necessarie al compimento del task obiettivo del flow. Ad ogni turno conversazionale si può verificare una transizione verso un'altra pagina (o anche no, in base all'interazione avuta). Ogni flow possiede almeno una pagina, cioè quella di partenza (denominata Start).

Un esempio di struttura di un flow è quello nella figura soprastante. In questo caso le pages sono i rettangoli blu, "Start" è la pagina iniziale e "Confirmation" rappresenta una transazione verso l'omonimo flow. Le frecce rappresentano le transizioni verso un'altra page (o un'altro flow, cioè verso la page Start di quel flow).

- Form: definisce i dati da raccogliere per una page (rappresenta una pre-condizione all'attivazione della page, quindi prima i dati vengono raccolti e poi viene invocata la page). Il concetto è simile a quello di slots degli altri assistenti. La raccolta avviene tramite un'interazione con l'utente potenzialmente in più turni di conversazione e prosegue fino a quando tutti i parametri richiesti non sono stati raccolti. Per ognuno di essi si possono/devono definire delle domande che verranno poste per chiedere il valore dei parametri all'utente.

- Fulfillment: al termine di un turno conversazionale, l'agente deve interagire con l'utente o terminare la sessione. Inoltre, potrebbe dover contattare un servizio esterno [webhook] per ottenere una risposta generata dinamicamente oppure per compiere un'azione (come salvare un appuntamento sul calendario). Queste azioni sono gestite dalla proprietà di fulfillment di una pagina. Nello specifico, il fulfillment può contenere una risposta statica (quindi una frase), una chiamata ad un webhook oppure un'istruzione di assegnazione od override di parametri. Dialogflow CX consente di definire più fulfillment, che verranno salvati in una coda (response queue). Al termine del turno conversazionale dell'agente, all'utente verranno restituite tutte le risposte in coda (in ordine).

- State Handler: controllano le informazioni di ogni stato e determinano se, ad esempio, ritornare un messaggio all'utente oppure far avvenire una transizione (cambio di stato). Sono un po' come dei "vigili urbani" che dirigono la conversazione, instradandola in direzioni diverse. Ci sono tre proprietà degli handler:

  – *requirements*, rappresentano i requisiti che devono essere soddisfatti affinché l'handler possa aver effetto (quindi possa essere chiamato)

  – *fulfillment*, parametro opzionale per definire una risposta all'utente (supporta anche la chiamata a webhook);

  – *transition target*, indica l'obiettivo dell'handler, ovvero la nuova page (una qualsiasi appartenente allo stesso flow, oppure la Start di un flow diverso) da rendere attiva.

  Gli handler possono essere di due tipi: routes (chiamati quando un intento viene invocato oppure una serie di variabili di contesto soddisfa una condizione precisa) ed event (chiamati quando si verifica un evento esterno, come il fallimento di una chiamata al webhook o un input inaspettato dell'utente). Il processo di invocazione di un handler si compone di tre step:

  1. *Scope*, ovvero viene controllato se il contesto attuale corrisponde a quello definito per l'handler; *Evaluation*, ovvero vengono controllati i requirements di ogni handler in scope; *Call*, che consiste nell'invocare l'handler se esso ha superato la fase di evaluation.

Le novità concettuali introdotte in Dialogflow CX si ripercuotono anche a livello dell'interfaccia di addestramento. Probabilmente la novità più grande rispetto a Dialogflow ES, ma anche rispetto a LEX e Watson, è la possibilità di visualizzare le pages e i collegamenti tra esse (quindi la struttura del dialogo) via flow-chart. Un esempio è fornito in figura 4.21.

Ad esempio in questa figura sono visibili due *pages*, "Store Location" e "Store Hours", collegate a "Start" tramite due route handlers.

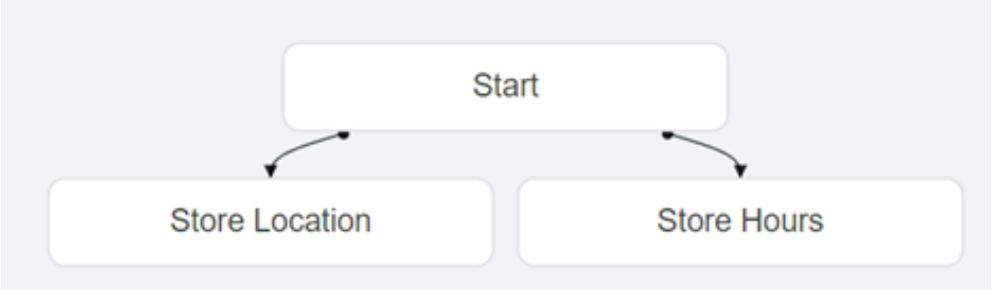


Figura 4.21: Flow-chart che visualizza pages e routes in Dialogflow CX

Premendo su una *page*, il rettangolo che la rappresenta si estende e mostra gli *handler* associati ad essa, rappresentati in figura 4.22.

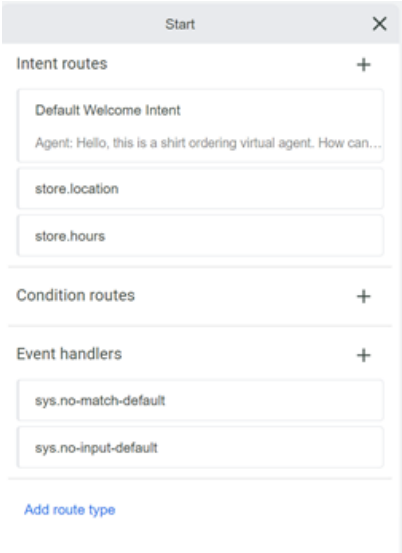


Figura 4.22: Dettaglio della visualizzazione degli handler in Dialogflow CX

Cliccando sopra ogni handler si apre un'altra finestra che permette di configurarne il comportamento. In base alla tipologia (*intent routes, condition routes* o *event*), le opzioni disponibili saranno differenti. Per un *intent route*, un esempio di schermata (riferito al "Default Welcome Intent) è riportato in figura 4.23.

Nella prima parte, tramite un menù a tendina, è possibile specificare quale sia l'intento associato all'handler. Il tasto "Edit Intent" rappresenta un collegamento rapido verso la schermata di modifica dell'intento selezionato. La voce "Condition" permette di definire delle condizioni di attivazione per l'handler tramite una logica if-then sui valori dei parametri della page corrente. La sezione "Fulfillment" permette di definire le azioni da compiere al termine dell'handler. Esse possono essere delle risposte testuali (che si possono inserire tramite un campo di testo), chiamate a webhook o preset di parametri. Infine, la sezione "Transition" permette facoltativamente di definire la transizione da scatenare al termine dell'esecuzione dell'handler.

La pagina di creazione e/o modifica di un intento, accessibile sia tramite il pulsante "Edit Intent" accennato in precedenza, sia tramite il menù di navigazione laterale è riportata in figura

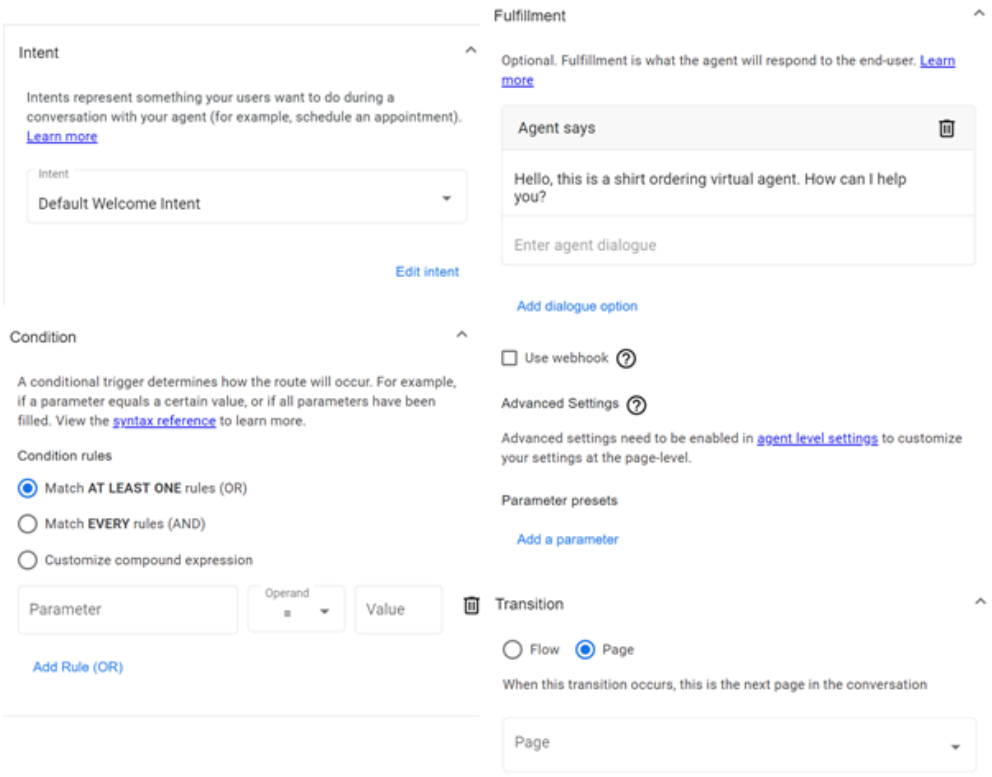


Figura 4.23: Configurazione del comportamento di un handler di tipo "intent route" in Dialogflow CX

4.24. Ogni intento ha un nome e delle frasi, che possono essere agevolmente inserite tramite casella di testo. L'interazione è analoga a quella di Dialogflow ES, LEX e Watson, quindi non presenta grosse peculiarità, se non la possibilità di marcare il riferimento ad un'entità all'interno di una delle *training phrases* semplicemente evidenziando la parola target.

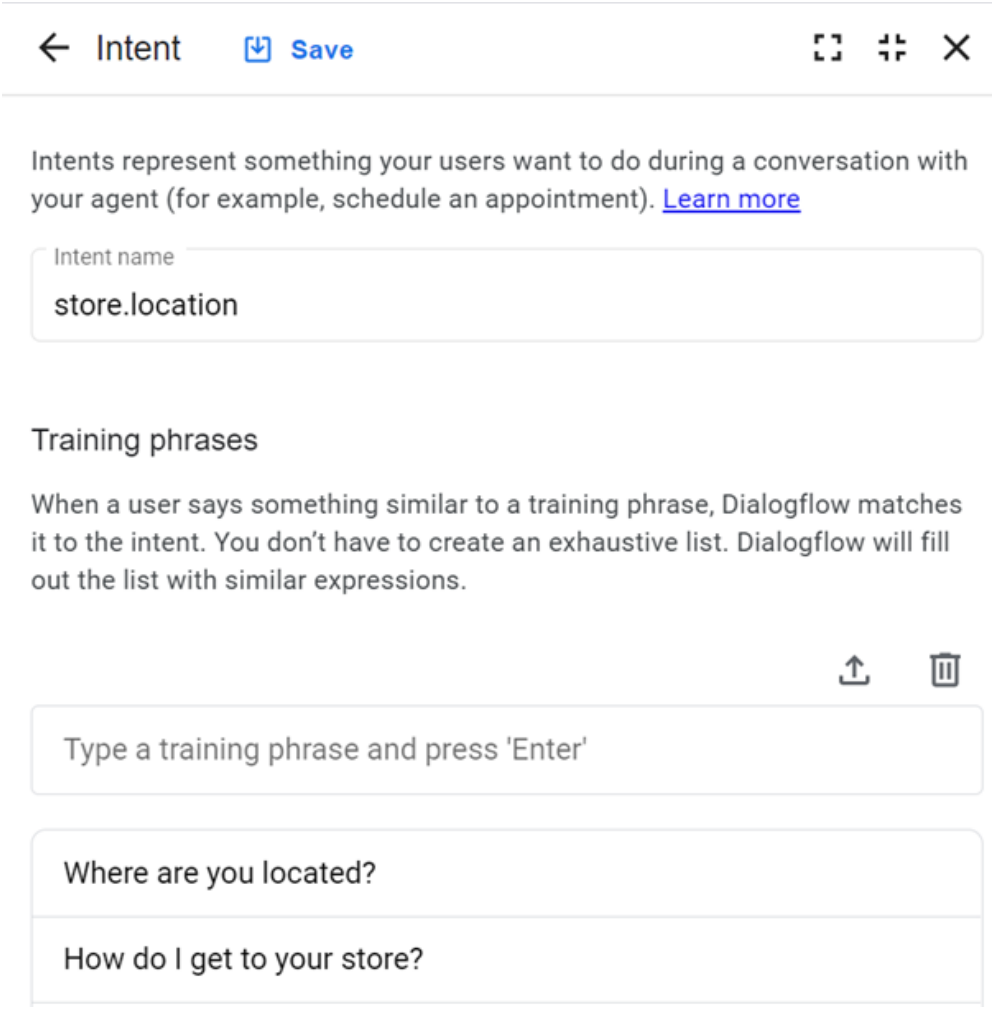


Figura 4.24: Modifica di un intento in Dialogflow CX

La pagina di modifica delle entità, accessibile dal menù laterale, è illustrata in figura 4.25. Come per il fratello Dialogflow ES, per ogni entità è necessario specificare un nome ed un elenco di valori che essa può assumere; per ogni ognuno di esso è caldamente suggerito inserire dei sinonimi. Un'ulteriore funzionalità è data dall'opzione "Automatically add entities", che permette di esaminare le entities già inserite e, sulla base dei loro valori, grazie al motore NLU della piattaforma, crearne di nuove in automatico.

4.2.3 IBM Watson

Parte della suite di servizi cognitivi IBM Watson, si prefigge di creare *chatbot* intelligenti (dotati di supporto NLP) in maniera molto rapida. Una caratteristica interessante è la mancata

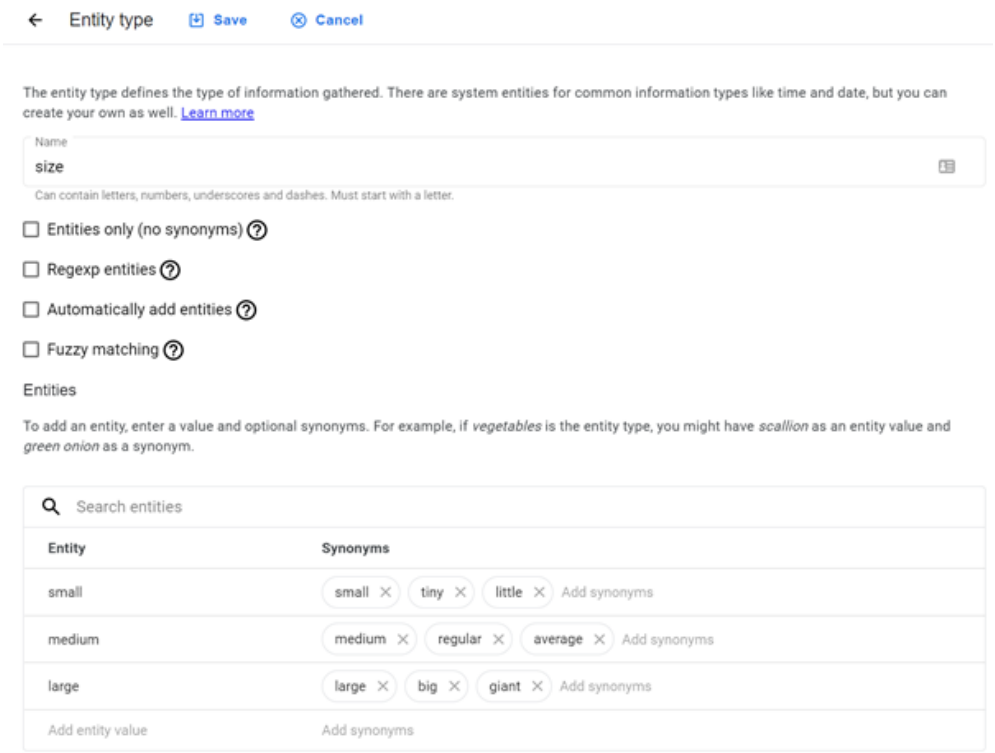


Figura 4.25: Modifica di un'entità in Dialogflow CX

distinzione tra interazione vocale e testuale: non importa come l'utente interagisce con il bot ma solo cosa dice. Questo si traduce nel non dover progettare due bot diversi, uno per tipologia di interazione. Come Amazon LEX e Dialogflow EX, anche Watson Assistant si basa sui concetti di intento, entità e dialogo.

- *Intento*: ogni intento è definito da un nome (che inizia con un hashtag #), una descrizione e una serie di esempi/frasi di training. Nelle frasi di training possono essere annotate le entità cliccando sopra la parola corrispondente: serve a far capire a Watson che la parola annotata rappresenta un'istanza di una determinata entità. La schermata di riferimento è riportata in figura 4.26.

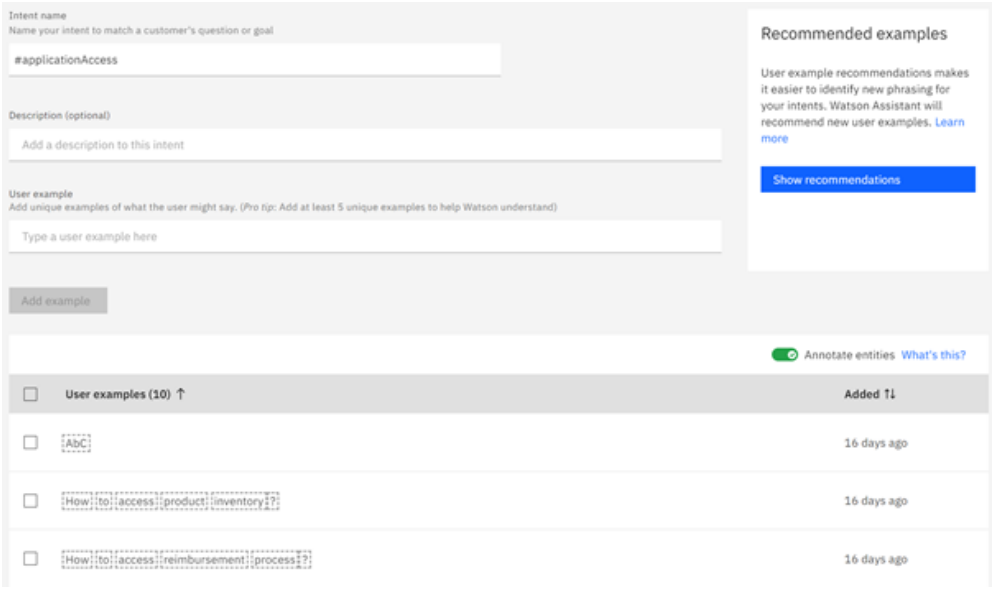


Figura 4.26: Modifica di un intento in IBM Watson

- *Entità*: ogni entità è definita da un nome (che inizia con una chiocciola @) e una lista di valori che può assumere. Ad ogni valore è possibile (anzi, opportuno) associare uno o più sinonimi, in modo da migliorare la precisione nel riconoscimento della stessa, come si può notare nella figura 4.27.

  Watson per le entità supporta anche il fuzzy matching, che permette di riconoscere le entità in modo meno "rigido", tollerando quindi errori di ortografia (leggeri).

- *Dialogo*: mentre Dialogflow EX associa la risposta del *chatbot* direttamente nell'intento, Watson definisce il comportamento in una schermata a parte, quella del dialogo appunto (riportata in figura 4.28). Essa usa un approccio visivo ad albero e permette di definire la logica di come comportarsi se e quando un particolare intento viene rilevato.

Ogni azione che il *chatbot* compie è rappresentata da un nodo, che in figura 4.28 è raffigurato con un rettangolino bianco. Ciascun nodo ha le seguenti proprietà (a cui si può avere accesso tramite il click sul rettangolo che lo rappresenta):

- *Nome*;

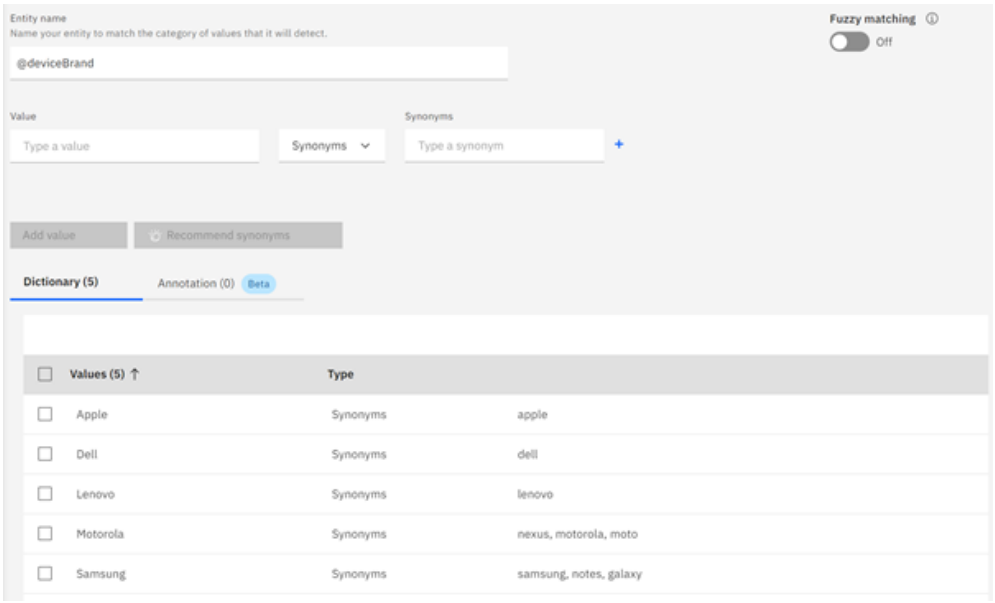


Figura 4.27: Modifica di un'entità in IBM Watson

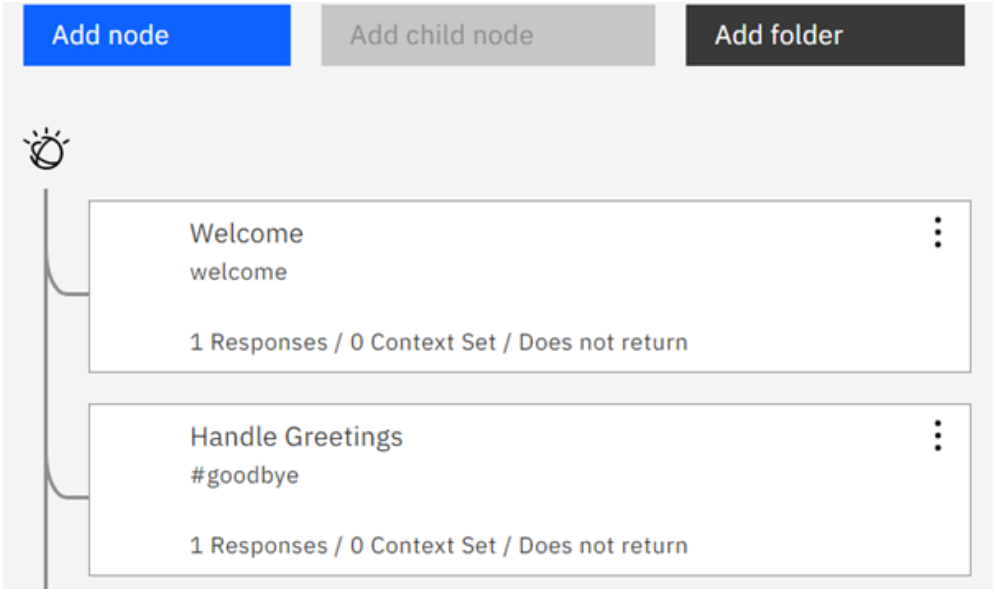


Figura 4.28: Visualizzazione del flusso conversazionale in IBM Watson

- *Intenti di attivazione*, che nella figura 4.29 è uno solo ed è indicato sotto la voce "If assistant recognizes". Questa è la lista di intenti che determina l'ingresso nel nodo corrente;
- *Azione da compiere al riconoscimento dell'intento* che, in base alla configurazione del nodo può essere:
  - Una semplice risposta testuale, magari estratta casualmente da una lista di possibili risposte (come in figura 4.29);
  - Una richiesta di informazioni aggiuntive: in tal caso apparirà la schermata presente in figura 4.31. In questa finestra è possibile definire l'elenco degli *slot* (cioè delle entità) che si intende valorizzare e, per ognuna di esse, il nome della variabile di contesto associata e il *prompt*, ovvero la frase che verrà utilizzata per richiedere all'utente quell'informazione
- L'*azione di fulfillment*, ovvero cosa deve fare il bot una volta che ha raccolto tutte le informazioni richieste. Le possibili azioni sono *Wait for reply* (ovvero rimane in attesa di una risposta da parte dell'utente) e *Jump to* (che consiste nello "spostare" il contesto del dialogo in un altro nodo).

Il comportamento del nodo, ovvero le azioni che l'assistente compie/la logica che applica al riconoscimento delle condizioni di ingresso (quindi quando il nodo viene "attivato") può essere ulteriormente personalizzata premendo sul pulsante "Customize" che si trova a destra del titolo.

Dalla schermata che appare (figura 4.30) è possibile specificare se:

- Il nodo necessita di alcune informazioni di input (slots) per poter elaborare la risposta. In tal caso, se l'informazione non è già stata raccolta, il *chatbot* provvederà a richiederla all'utente e a valorizzare la relativa variabile di contesto;
- Il nodo deve gestire più opzioni di risposta in base ad alcune condizioni (ad esempio, al valore delle variabili di contesto);
- Il nodo dovrà interagire con un servizio esterno tramite chiamata API per recuperare dei dati (webook)

La figura 4.32 illustra una sezione della schermata di dettaglio del nodo (stesso tipo della figura 4.30) che appare quando per quel nodo viene attivata la possibilità di avere risposte multiple condizionate. Qui viene applicata una logica del tipo "if-then" che, al verificarsi della condizione specificata nella colonna "If assistant recognizes" (che nel caso della prima riga si traduce con "se l'entità deviceType è valorizzata con 'tablet'"), produce in output la risposta scritta in "Respond with" ("Hi, i'm IT Support *chatbot*,...." nel caso della prima condizione).

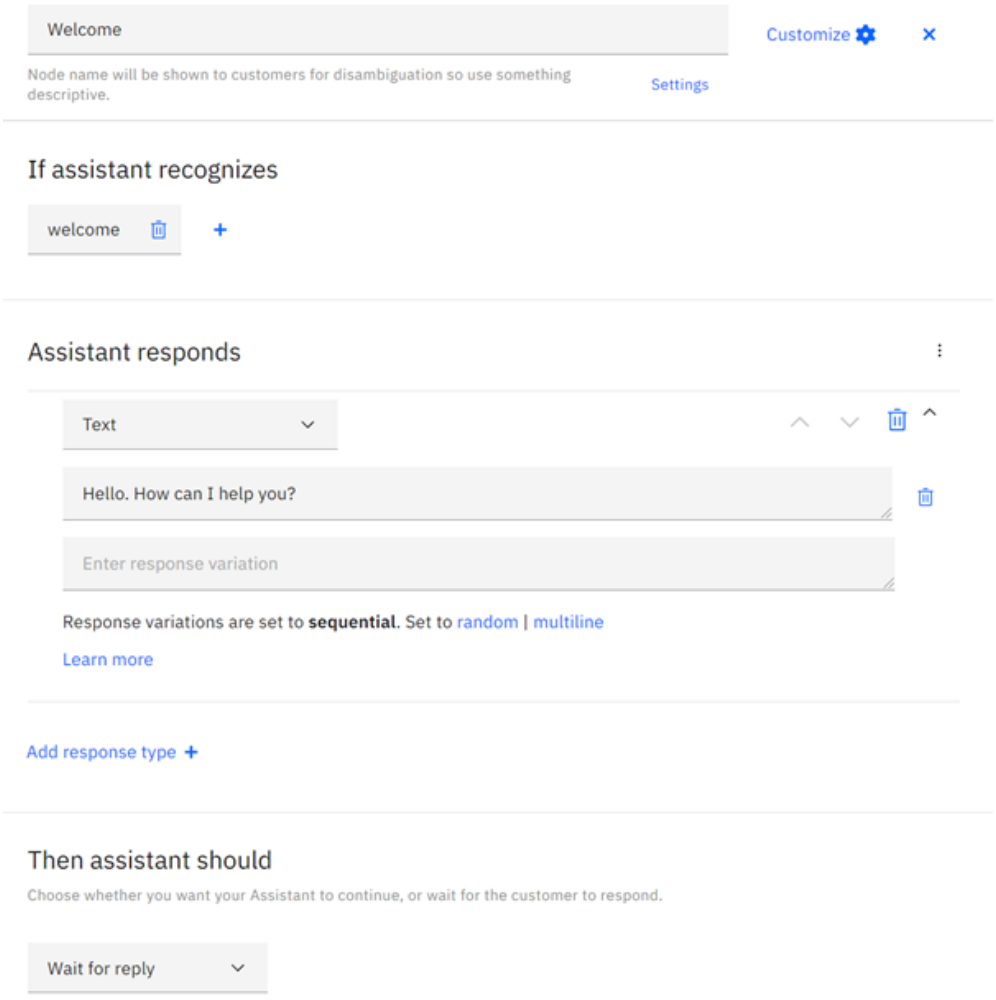


Figura 4.29: Definizione dell'azione di risposta ad un intento in IBM Watson

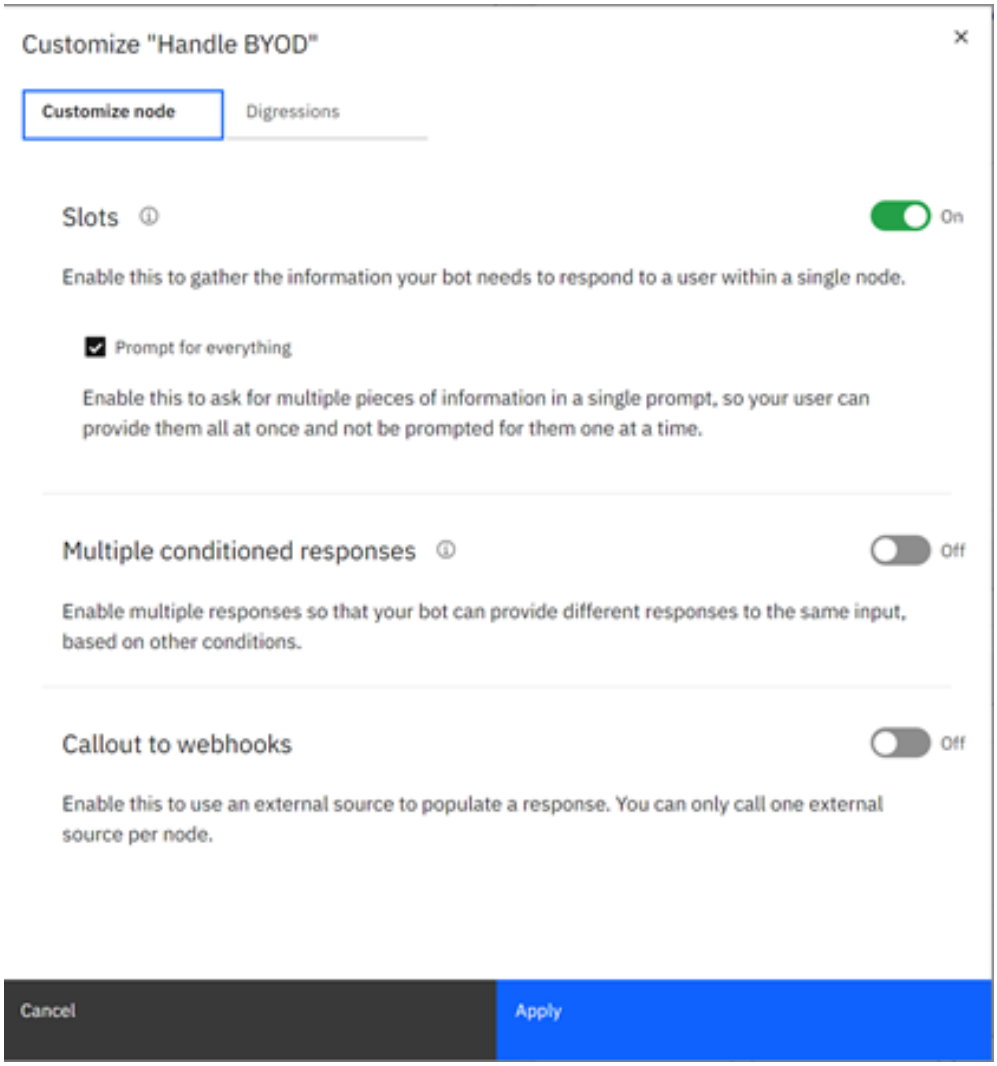


Figura 4.30: Dettaglio del comportamento di un nodo nel flusso conversazionale di IBM Watson

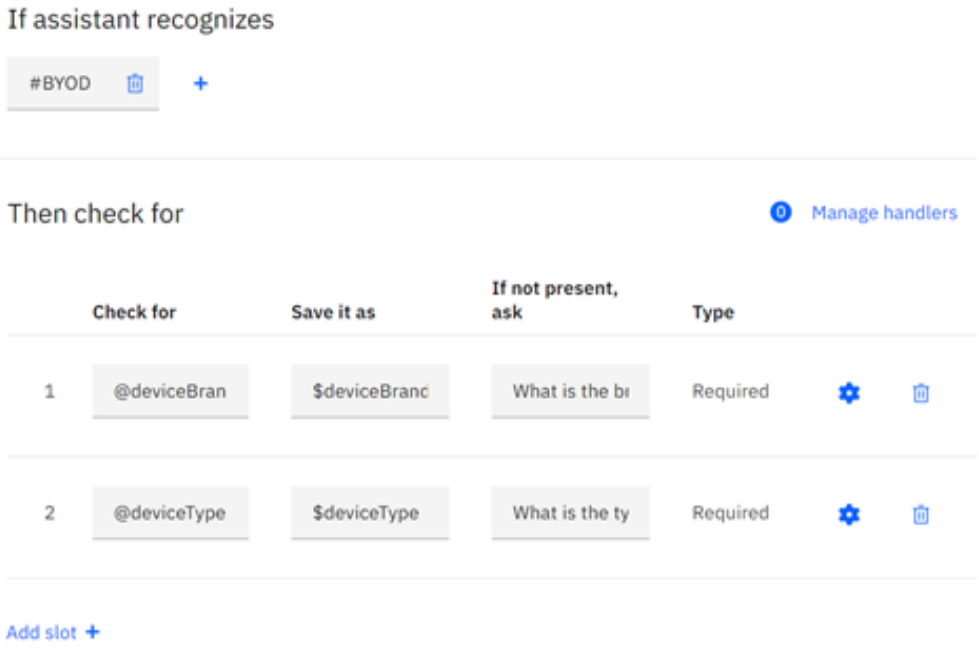


Figura 4.31: Definizione del comportamento di un nodo tramite slots in IBM Watson

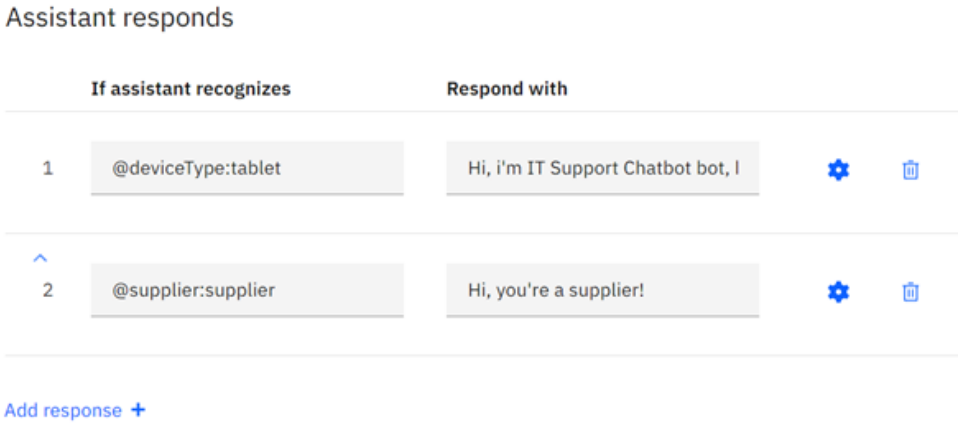


Figura 4.32: Dettaglio della definizione di una risposta multipla in IBM Watson

Capitolo 5

Esame del *mockup* fornito dall'azienda

Alla luce dell'esperienza acquisita con l'analisi riportata in sezione 4.1 e 4.2 di numerose piattaforme di addestramento *chatbot* presenti sul mercato, l'attività successiva è stata l'esame di un prototipo di applicazione web, piattaforma di addestramento *chatbot* creato dall'azienda con Adobe XD.

5.1 Caratteristiche dell'applicazione

Il prototipo fornito rappresenta la visione dell'azienda di un ipotetico tutorial che porti un utente inesperto a creare il suo primo *chatbot*. A riguardo è necessario fare delle precisazioni in merito al tipo di piattaforma che l'azienda intende creare, al profilo dell'utente target e alle funzionalità che la piattaforma deve o non deve includere.

Tipo di piattaforma La piattaforma deve essere uno strumento che consenta di realizzare un *chatbot* intelligente, ovvero un *chatbot* che faccia uso di NLP / NLU . La piattaforma si pone come target delle realtà aziendali, dove le tipologie di di *chatbot* più frequenti sono i *FAQ chatbot*.

Profilo dell'utente L'utente target è una persona senza competenze informatiche specifiche, ovvero senza esperienza nel mondo dei *chatbot* e nel mondo della programmazione in generale. Idealmente, un esempio di utente potrebbe essere il responsabile marketing di un'azienda.

Funzionalità offerte La piattaforma deve permettere di:

- Creare un nuovo *chatbot*;
- Visualizzare e modificare la lista dei *chatbot* creati;
- Prevedere un meccanismo che faciliti la realizzazione di un FAQ *chatbot*, come l'upload di una *knowledge base* già esistente sotto forma di file "in stile Excel" (.xls, .xlsx o .csv);
- Caricare un *chatbot* creato su una piattaforma generalista (cioè una di quelle descritte in sezione 4.2) da definire.

Dall'altro canto, l'applicazione non si pone come scopo quello di realizzare un proprio motore di NLP , pertanto tutta la parte di intelligenza artificiale deve essere gestita dalla piattaforma generalista di appoggio (Amazon Lex, Google DialogFlow o IBM Watson).

5.2 Convenzioni adottate

Il primo problema che l'azienda ha dovuto affrontare è stato quello di trovare un vocabolario "non tecnico" per descrivere gli elementi caratteristici di un *chatbot*. A riguardo l'azienda, per la realizzazione del *mockup*, ha adottato le seguenti convenzioni:

Bisogno indica il problema che una lista di domande concorre alla risoluzione. Facendo un esempio, date le domande "Che tempo fa domani a Padova?" e "Quali sono le previsioni di domani per Padova?", entrambe sono specchio della stessa situazione di necessità: l'utente vuole sapere il meteo di Padova previsto per domani. Ecco che le domande possono essere raggruppate in un unico bisogno chiamato, ad esempio, "meteo di padova". Si può notare come il concetto di bisogno sia analogo a quello di intento.

Un bisogno possiede le seguenti caratteristiche:

- *Nome*, che non presenta particolari vincoli sintattici;
- Lista di *espressioni* (domande) che l'utente potrebbe utilizzare per esprimere la propria necessità. Alcune piattaforme analizzate in precedenza chiamano questa lista *sample utterances*;
- *Risposta* che il *chatbot* deve fornire per adempiere alla richiesta dell'utente. Esistono tre tipologie di risposta:
  1. *Testo semplice*, cioè una frase "secca";
  2. *Elemento multimediale*, come un'immagine, un video, un link ad una pagina web;
  3. *Dipende da*, ovvero una risposta dinamica che verrà "calcolata" in base a delle informazioni fornite dall'utente. Facendo un'analogia con il mondo della programmazione, questo tipo di risposta corrisponde ad un'istruzione *if-then-else*.

Tema indica un raggruppamento di bisogni di uno stesso ambito. L'idea dell'azienda è di utilizzare i temi per "mettere ordine" tra i bisogni. Ad esempio, se viene creato un *FAQ chatbot* con un centinaio di bisogni, essi possono essere raggruppati in base alla tematica che affrontano assegnandoli allo stesso tema, in modo da poter poi trovarli in maniera più rapida.

5.3 Analisi del flusso

Come già anticipato, il *mockup* realizzato dall'azienda è stato fornito sotto forma di flusso di interazione realizzato con Adobe XD. Tale flusso si compone di diverse schermate, che corrispondono alle possibili interazioni che l'utente può svolgere. Ciascuna schermata verrà analizzata più nel dettaglio nelle sezioni che seguono.

5.3.1 Homepage

Lo scopo della schermata di homepage riportata in figura 5.1 è quello di visualizzare la lista dei *chatbot* esistenti e di permettere all'utente, tramite la pressione del tasto "+", di crearne uno

Figura 5.1: Homepage del *mockup* fornito dall'azienda

nuovo. Premendo sul bottone "+" esce una finestra di dialogo che consente di scegliere se si vuole effettuare la procedura guidata, che è quella illustrata nel flusso fornito dall'azienda, o se invece saltare la guida e proseguire in maniera più "libera".

Una proposta di modifica sottoposta all'azienda e anche accettata è stata di rendere la guida "intelligente", ovvero fare in modo che:

- la guida compaia in automatico solo la prima volta che l'utente utilizza la piattaforma;
- la guida sia richiamabile da ogni schermata dell'applicazione (tramite un tasto con l'icona del "?") e sia contestuale.

5.3.2 Identità del bot

L'azienda ha pensato di dedicare uno step del processo di creazione del bot alla definizione della sua "identità". In questa schermata, riportata in figura 5.2, viene richiesto di esplicitare il nome del *chatbot* che si sta creando e, facoltativamente, una foto/icona che ne faciliti l'identificazione. Inoltre sul lato sinistro di questa schermata sono presenti due elementi: la guida, che compare sotto forma di popup, e un indicatore di avanzamento nel processo di addestramento. Entrambi questi elementi sono decisamente utili e si è deciso di mantenerli.

Infine un ultimo particolare degno di nota è il colore "primario", ovvero quello utilizzato come sfondo della barra del titolo e del pulsante con etichetta "continua". Si è deciso infatti di adottare un colore primario diverso per ogni caratteristica del *chatbot*: l'identità utilizzerà l'arancione, il bisogno il giallo, le domande il verde, il tema il verde acqua e così via. Questo serve anche a "compartimentare" a livello logico la procedura di addestramento, rendendola meno caotica.

5.3.3 Inserimento delle domande

Per la creazione del primo bisogno, l'azienda ha scelto un approccio "rovescio": invece di inserire prima i dettagli del bisogno e poi le frasi/domande trigger, viene fatto l'esatto contrario.

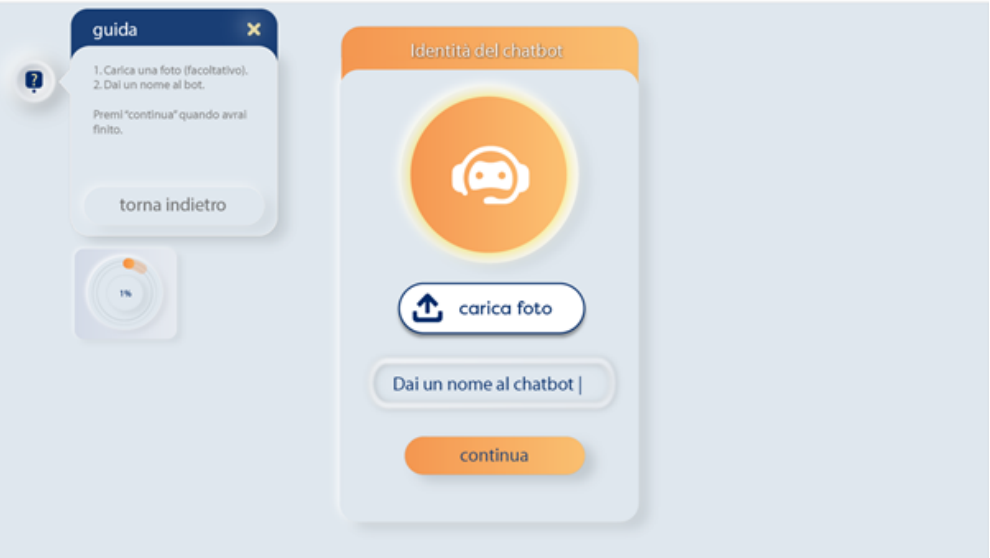


Figura 5.2: Selezione dell'identità del *chatbot* nel *mockup* fornito dall'azienda

La schermata che appare in figura 5.3 è quella che viene proposta in seguito a quella in figura 5.2. Oltre alla guida e alla barra di avanzamento appaiono due sezioni:

- "La prima domanda", in cui è possibile inserire manualmente la prima *utterance* da associare al primo bisogno;
- una sezione che consente di caricare un file strutturato con una *knowledge base* esistente. Il file poi dovrà essere interpretato dal sistema e l'idea è che vengano estrapolati i bisogni con la lista delle domande e la risposta in maniera automatica. Questa funzionalità è stata ritenuta fondamentale per velocizzare la realizzazione di FAQ *chatbot*.

Il *mockup* poi illustra un prototipo di schermata di inserimento manuale delle domande (riportata in figura 5.4, quindi quella che dovrebbe comparire quando si preme il tasto "Continua" in figura 5.3.

In questa schermata le domande dovrebbero apparire una sotto l'altra man mano che vengono inserite, con la possibilità anche di eliminarle premendo sulla "x" di fianco al testo. Anche se non è presente nessun pulsante o icona che lo indica, per modificare una domanda è sufficiente cliccarci sopra: il suo contenuto verrà copiato sulla casella di testo e sarà possibile editarlo.

Nella parte centrale è visibile una rappresentazione della base di conoscenza del *chatbot* sotto forma di insieme (l'azienda, per descriverla, ha usato un'analogia con i "barbapapà"). L'idea di fondo è che questi insiemi, una volta popolati, possano essere collassati ed espansi in modo da permettere un'esplorazione visuale ordinata e veloce della base di conoscenza del *chatbot*.

5.3.4 Inserimento del nome del bisogno

Una volta definita la lista delle domande, viene richiesto di inserire il nome del bisogno tramite la schermata in figura 5.5.

Si noti come la rappresentazione insiemistica della base di conoscenza (nella parte centrale) inizi a prendere forma, con il nome del bisogno che viene riportato su sfondo giallo sopra ai

Figura 5.3: Inserimento della prima domanda nel *mockup* fornito dall'azienda

Figura 5.4: Inserimento della lista delle domande nel *mockup* fornito dall'azienda

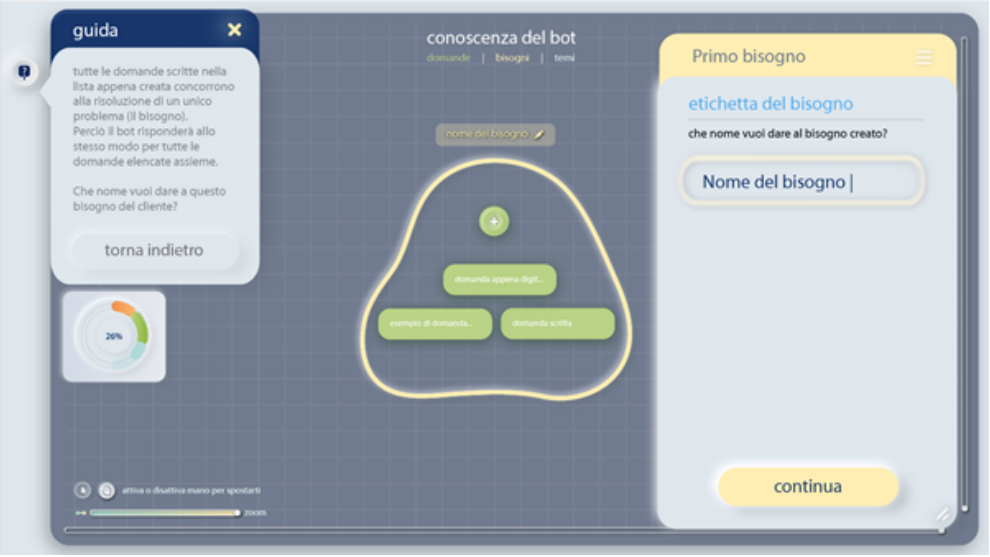


Figura 5.5: Inserimento del nome del bisogno nel *mockup* fornito dall'azienda

confini dell'insieme. A livello di modifiche, una proposta fatta all'azienda (e accolta) è stata pre-compilare il nome del bisogno con il testo di una delle domande inserite, in modo da aiutare l'utente inesperto nella comprensione del concetto stesso di bisogno.

5.3.5 Definizione del tema

In seguito all'inserimento del nome del bisogno viene richiesto all'utente il nome del tema tramite la schermata riportata in figura 5.6.

5.3.6 Risposta del bisogno

Lo step successivo (figura 5.7) consiste nell'inserimento della risposta del bisogno corrente. La risposta può essere di tre tipi: "testo semplice", "lineare" (cioè un elemento multimediale o un link) oppure di tipo "dipende da" (che risulta anche selezionato di default).

5.3.6.0.1 Dipende da In caso di risposta di tipo "dipende da", all'utente viene presentata una schermata (riportata in figura 5.8) che consente di creare in maniera visuale il diagramma di flusso alla base del ragionamento. Premendo su uno dei due pulsanti "+" sotto le due opzioni di risposta ("si"/"no") compare un menù a tendina che consente di aggiungere una risposta a quel ramo del diagramma. Ancora una volta, la risposta che si può aggiungere potrà essere del testo semplice, un elemento multimediale o un'altra dipendenza. Facendo un paragone con le piattaforme "generaliste", ogni blocco di "dipende da" corrisponde ad uno *slot* e le opzioni di risposta definiscono il dominio di quello slot, ovvero l'entità ad esso associata. Il tutto però avviene senza che l'utente debba preoccuparsi di definire a priori le entità, creare delle variabili di sessione o, più in generale, pensare "da informatico".

5.3.6.0.2 Risposta testuale o lineare Per inserire un'opzione di risposta testuale o lineare la schermata proposta è quella riportata in figura 5.9. In caso venga selezionato questo tipo di

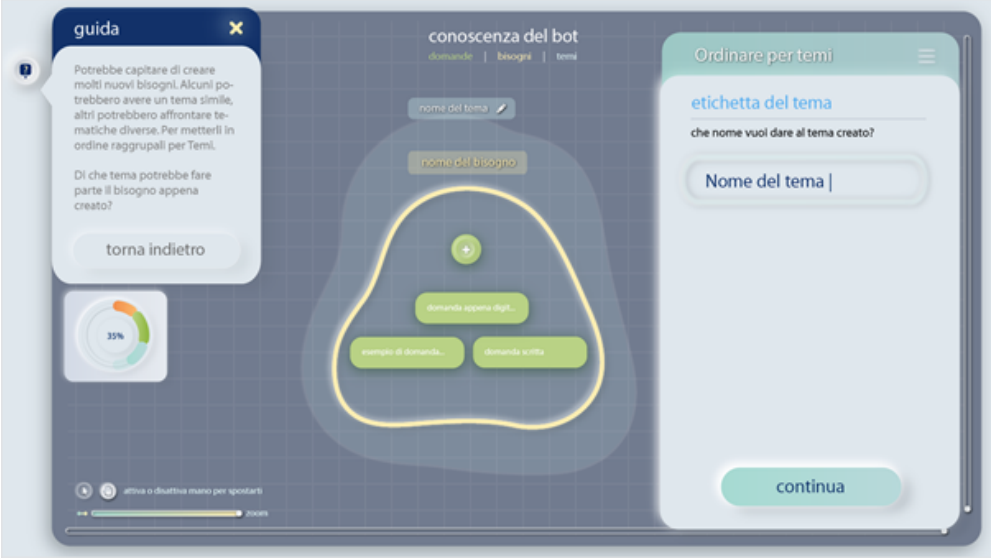


Figura 5.6: Inserimento del nome del tema nel *mockup* fornito dall'azienda

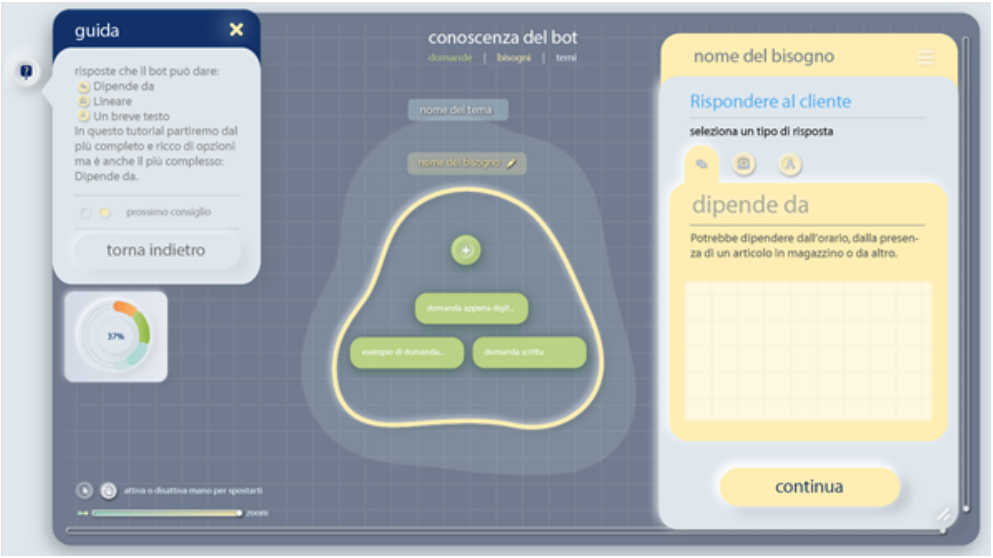


Figura 5.7: Selezione della risposta al bisogno nel *mockup* fornito dall'azienda

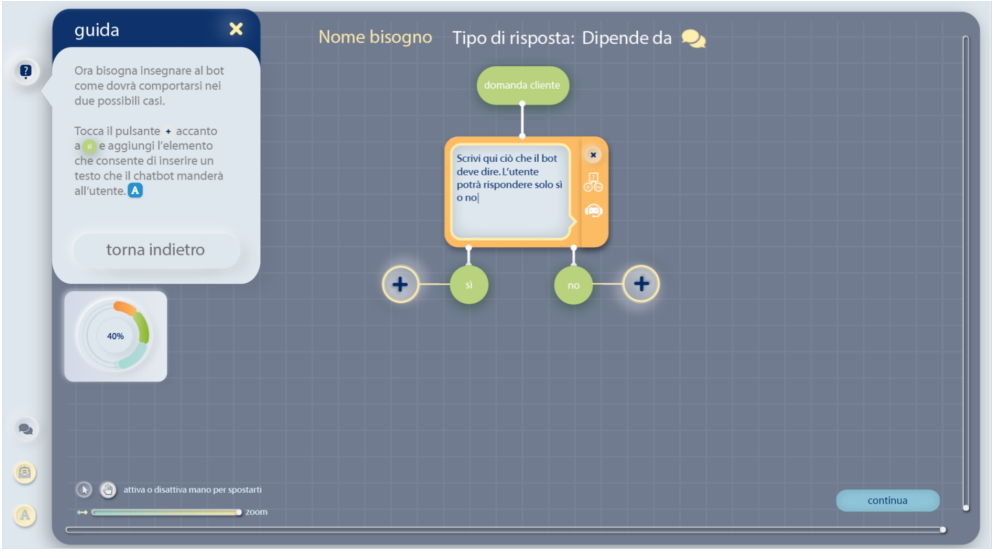


Figura 5.8: Inserimento risposta di tipo "dipende da" nel *mockup* fornito dall'azienda

Figura 5.9: Inserimento risposta di tipo testuale nel *mockup* fornito dall'azienda

risposta, il *chatbot*, una volta riconosciuto il bisogno, dovrà restituirla direttamente all'utente, senza bisogno di richiedere input aggiuntivi, come invece avviene con il "dipende da".

5.3.7 Test del *chatbot*

Una volta completato l'inserimento di tutte le informazioni necessarie, viene proposta una sezione in cui è possibile testare il comportamento del *chatbot*, come riportato in figura 5.10. L'idea è rendere possibile all'utente di interagire con il prodotto appena creato per poterlo

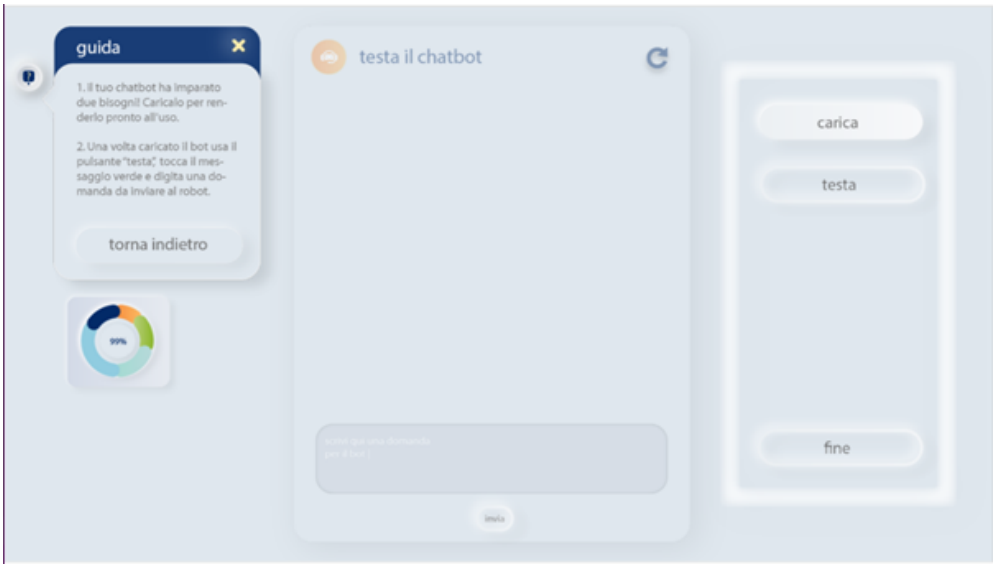


Figura 5.10: Test del *chatbot* appena creato nel *mockup* fornito dall'azienda.

testare in modo rapido, prima di effettuare il deploy finale. Tuttavia, dopo un'attenta analisi di fattibilità, è stato deciso di non realizzare questa sezione poiché, sebbene possa risultare utile, richiederebbe l'implementazione di un motore di NLP proprietario, caratteristica che non appartiene allo scopo dell'applicazione.

5.3.8 Riepilogo finale

Al termine dell'addestramento viene visualizzata la schermata di riepilogo riportata in figura 5.11. Gli elementi presenti sono:

- L' immagine identificativa del *chatbot* inserita all'inizio;
- Un punteggio da 1 a 5 stelle, la cui idea è di indicare il grado di "completezza" del *chatbot* appena addestrato. Dopo una discussione con l'azienda, tuttavia, non si è riusciti ad individuare una metrica affidabile per l'attribuzione del punteggio, quindi si è deciso di rimuovere questo elemento;
- Due tasti, "modifica" e "pubblica". Quest'ultimo dovrebbe permettere di caricare in maniera (semi)automatica il *chatbot* creato nella piattaforma generalista scelta.

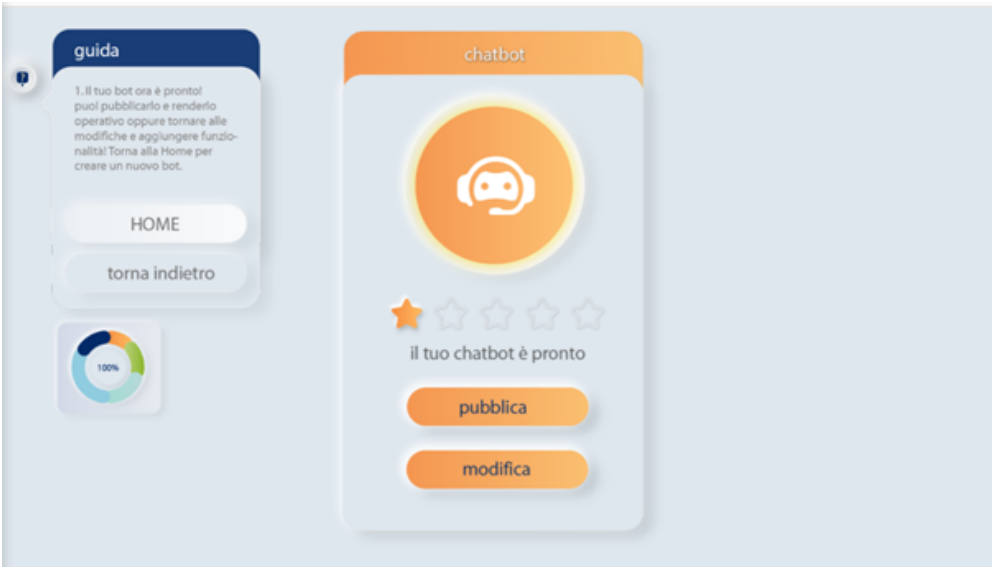


Figura 5.11: Schermata di riepilogo del *chatbot* appena creato nel *mockup* fornito dall'azienda.

5.4 Esempio pratico

Lo scopo di questa sezione è illustrare il funzionamento della procedura di inserimento di un bisogno secondo il *mockup* fornito dall'azienda tramite un esempio pratico. Poniamo esista una pizzeria che vuole realizzare un *chatbot* capace di fornire un preventivo del costo di consegna di una pizza, il quale viene così calcolato:

- Se il cliente abita a Padova, il costo di consegna è di 1 euro;
- In caso contrario, la consegna costa 3 euro.

La pizzeria è piccola, a conduzione familiare e nessuno ha competenze informatiche: per questo motivo decidono di affidarsi alla piattaforma.

Una volta aperta la piattaforma, l'utente si troverà davanti alla schermata riportata in figura 5.1, cioè all'homepage dell'applicazione. Volendo creare un nuovo *chatbot* premerà sul pulsante "+", il quale causerà l'avvio del tutorial.

Lo step successivo è l'inserimento dell'identità del *chatbot*, quindi immagine e nome. Supponiamo di chiamare il *chatbot* "Pizzeria Al Campanile" inserendo il nome nell'apposito campo di testo, senza nessuna immagine. La schermata che ne risulterà sarà quella in figura 5.12. Terminato l'inserimento del nome, l'utente dovrà premere il tasto *continua*, che lo porterà nella schermata di inserimento della prima domanda (cioè quella della figura 5.3). Come anche suggerito dalla guida, l'utente proverà a pensare ai modi in cui i suoi clienti potrebbero rivolgersi al *chatbot*, giungendo alle seguenti espressioni:

- *Quanto costa la consegna?*
- *A quanto ammontano le spese di consegna?*

Deciderà di inserire "Quanto costa la consegna?" nel campo di testo, poiché è la prima espressione tra le due che gli è venuta in mente. Al termine dell'interazione, la schermata visualizzata

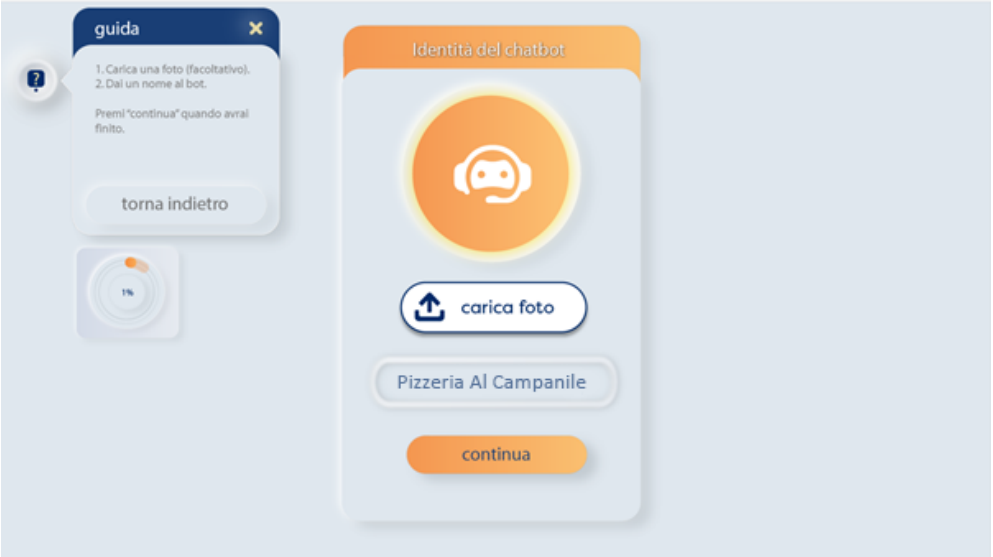


Figura 5.12: Schermata "identità" del *chatbot* "Pizzeria Al Campanile"

Figura 5.13: Schermata "prima domanda" del *chatbot* "Pizzeria Al Campanile"

sarà quella della figura 5.13. Utilizzando ancora una volta il tasto "continua", l'utente verrà condotto allo step che chiederà di inserire eventuali altri modi per formulare la stessa domanda, come nella schermata 5.4. Siccome erano state individuate due espressioni con cui un cliente avrebbe potuto rivolgersi al *chatbot*, ora l'utente inserirà anche la seconda, cioè "A quanto ammontano le spese di consegna?". Per farlo utilizzerà il campo di testo presente nella parte destra della schermata e, terminato l'inserimento, premerà sul tasto "continua". Una ricostruzione di quanto si troverà davanti è riportata in figura 5.14. Un particolare da notare è che nella sezione centrale della schermata la rappresentazione "a barbapapà" inizia a popolarsi, contenendo al momento solo le due domande precedentemente inserite.

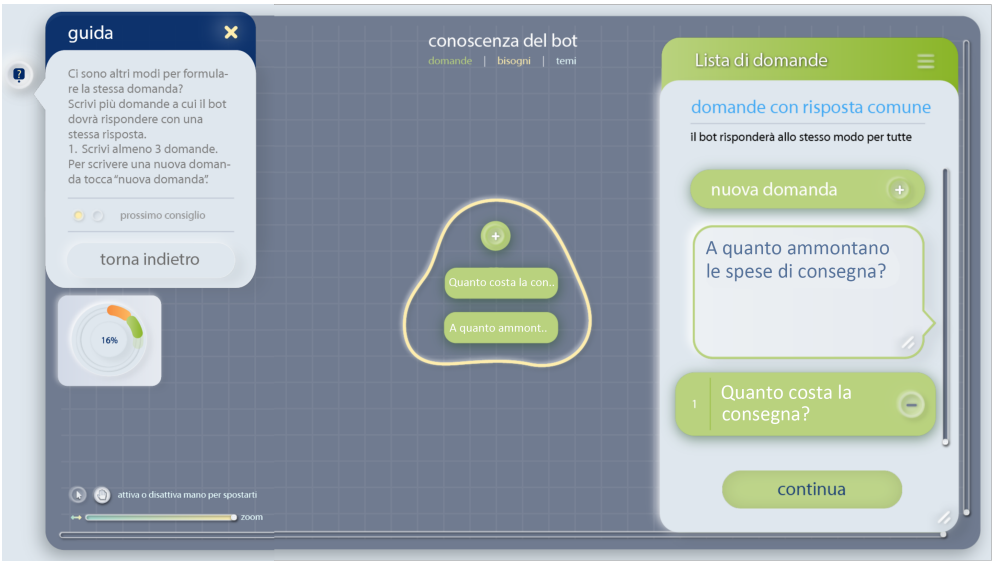


Figura 5.14: Schermata "Lista Domande" del *chatbot* "Pizzeria Al Campanile"

Al termine dell'inserimento della domanda, l'utente premerà sul tasto "continua" e verrà condotto alla schermata di inserimento del nome del bisogno (figura 5.5). La guida introduce l'utente al concetto di bisogno con la seguente frase: *"Tutte le domande scritte nella lista appena creata concorrono alla risoluzione di un unico problema (il bisogno). Perciò il bot risponderà allo stesso modo per tute le domande elencate assieme. Che nome vuoi dare a questo bisogno?"*. Dopo un po' di riflessione, l'utente decide di chiamare il bisogno semplicemente "consegna", visto che si occupa del calcolo del costo di consegna a domicilio. Per farlo dovrà inserire la parola "consegna" nel campo di testo presente sulla destra, come mostrato nella figura 5.15. Da notare anche qua l'aggiornamento della rappresentazione insiemistica presente al centro della schermata, che riporterà sopra l'elenco delle domande la scritta "consegna" in giallo, per ribadire che quelle domande si riferiscono al suddetto bisogno.

Completato anche questo passaggio, l'utente dovrà premere sul tasto "continua", che lo condurrà all'inserimento del nome del tema. La guida contestuale presente nel lato sinistro della schermata (riportata in figura 5.6) recita: *"Potrebbe capitare di creare molti nuovi bisogni. Alcuni potrebbero avere un tema simile, altri potrebbero affrontare tematiche diverse. Per metterli in ordine raggruppali in Temi. Di che tema potrebbe fare parte il bisogno appena creato?"*. Siccome il bisogno riguarda una richiesta di informazione in merito ai servizi offerti dalla pizzeria, l'utente decide di chiamare il tema "servizi". Per farlo dovrà digitare il nome sulla casella di testo presente nel lato destro della schermata. Questo provocherà anche un aggiornamento della rappresentazione insiemistica presente al centro, come mostrato nella

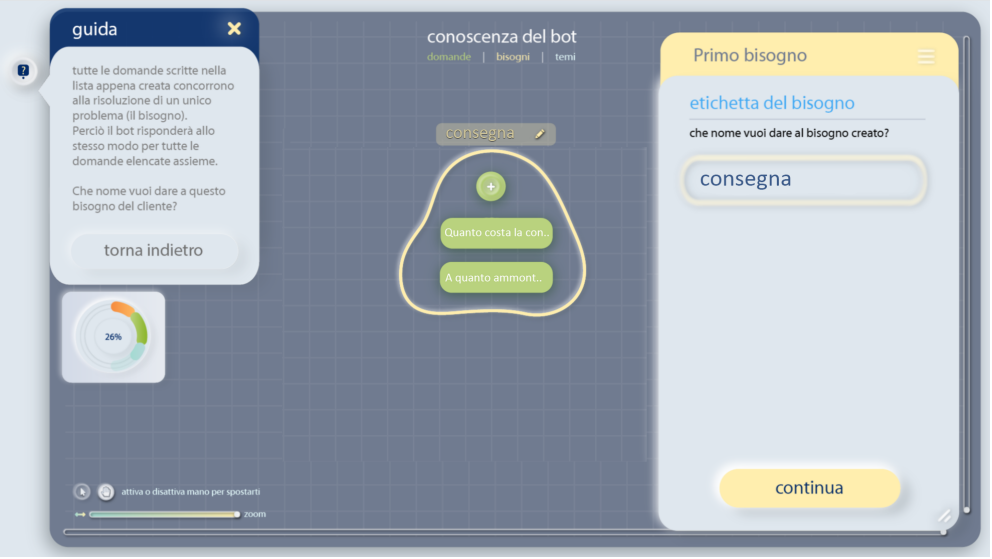


Figura 5.15: Schermata "Nome del bisogno" del *chatbot* "Pizzeria Al Campanile"

figura 5.16.

Premendo sul tasto continua, all'utente verrà proposto di inserire la risposta al bisogno tramite la schermata della figura 5.17. La guida contestuale in questo caso recita: *"Quando un cliente farà una domanda simile a quelle che hai scritto, il robot riconoscerà di che tema si tratta e che bisogno specifico dovrà risolvere. A questo punto devi decidere che tipo di risposta dovrà dare al cliente. Ci sono 3 tipi di risposte che il bot può dare:*

- *Dipende da;*
- *Lineare;*
- *Un breve testo."*

Inoltre, nella parte sinistra della schermata viene pre-selezionato il tipo di risposta "Dipende da", la cui descrizione riporta: *"Potrebbe dipendere dall'orario, dalla presenza di un articolo in magazzino o da altro"*. Siccome la risposta che dovrà dare il *chatbot* dipenderà dal luogo di residenza del cliente (Padova o meno), il tipo di risposta "dipende da" sembra quello più indicato. Pertanto l'utente dovrà limitarsi a premere il tasto "continua", che lo condurrà alla schermata di definizione del ragionamento utilizzato per rispondere, corrispondente a quella riportata in figura 5.8.

Qui, come suggerito dalla guida, premendo sul tasto "+", si aprirà un menu a tendina che consentirà di scegliere il tipo di nodo da inserire nel diagramma. Come suggerito dalla guida contestuale verrà selezionata l'opzione "dipende da: risposta più Si o No", in quanto il ragionamento si basa su una scelta "booleana", cioè sul fatto che il cliente sia di Padova o meno. Il tutto è riportato in figura 5.18.

All'interno della casella di testo del nodo l'utente, come suggerito dal *placeholder*, dovrà inserire la domanda che il bot porrà all'interlocutore, cioè *"Sei di Padova?"* (figura 5.19a). Successivamente sarà necessario inserire le risposte in base alla scelta del cliente, come mostrato in figura 5.19b.

Al termine della definizione della logica di risposta, l'utente dovrà premere sul tasto "continua" in basso a destra,

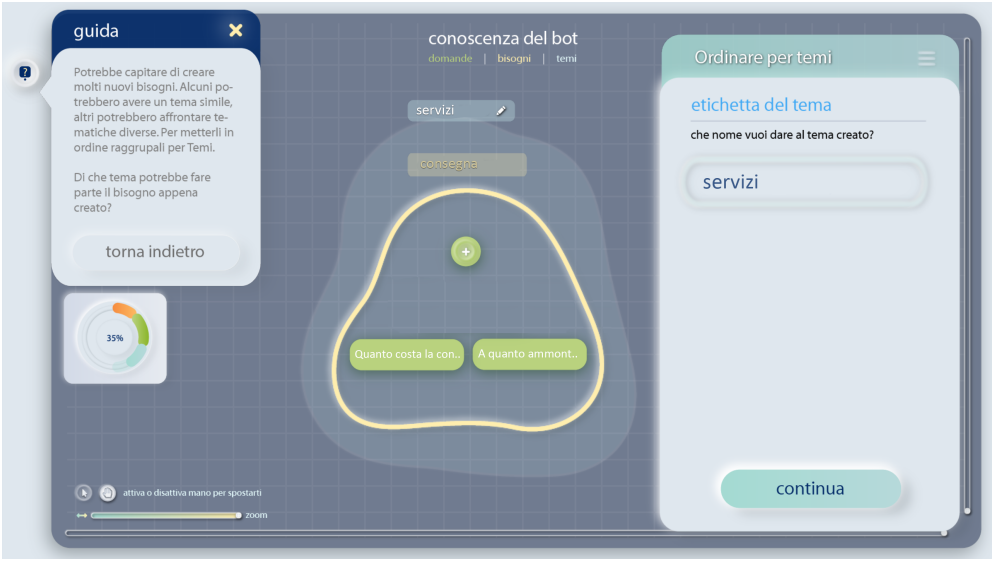


Figura 5.16: Schermata "Nome del tema" del *chatbot* "Pizzeria Al Campanile"

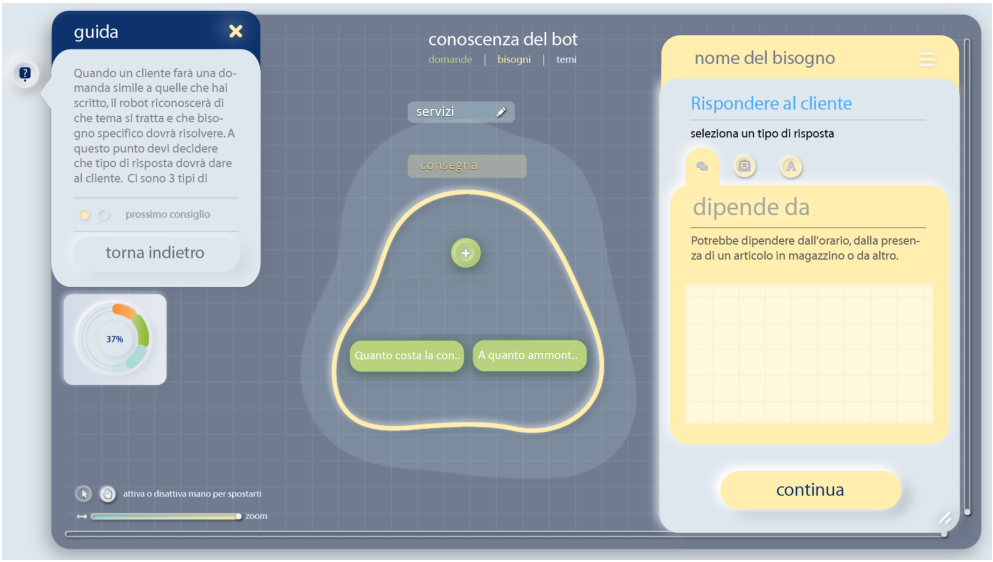
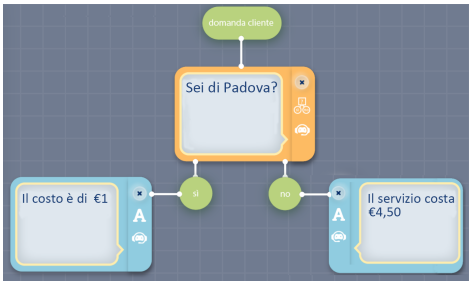


Figura 5.17: Schermata "Rispondere al cliente" del *chatbot* "Pizzeria Al Campanile"

Figura 5.18: Schermata "Dipende da" del *chatbot* "Pizzeria Al Campanile"

(a) Inserimento del testo della domanda da porre al cliente



(b) Inserimento delle due risposte

Figura 5.19: Dettaglio della creazione di una risposta di tipo "dipende da"

Capitolo 6

Progettazione della soluzione

Scopo di questo capitolo è illustrare le scelte che sono state fatte durante la fase di progettazione. Esse hanno riguardato sia l'interfaccia grafica ché lo stack tecnologico da adottare.

6.1 Piattaforma generalista di supporto

La prima scelta che si è dovuta fare è stata la scelta della piattaforma generalista in cui esportare il *chatbot* creato con l'applicazione. Le opzioni esaminate sono state quelle descritte in sezione 4.2, quindi *Amazon Lex*, *IBM Watson* e *Google DialogFlow* (in entrambe le versioni, ES e CX). Le considerazioni vengono riportate nei paragrafi successivi.

Google Dialogflow Al momento della fase di progettazione DialogFlow CX, cioè la versione più completa, era appena stata introdotta sul mercato ed era ancora in fase di beta. Nella documentazione [40] era presente il warning riportato in figura 6.1, che non ne garantiva la stabilità delle funzionalità in quanto versione pre-release.

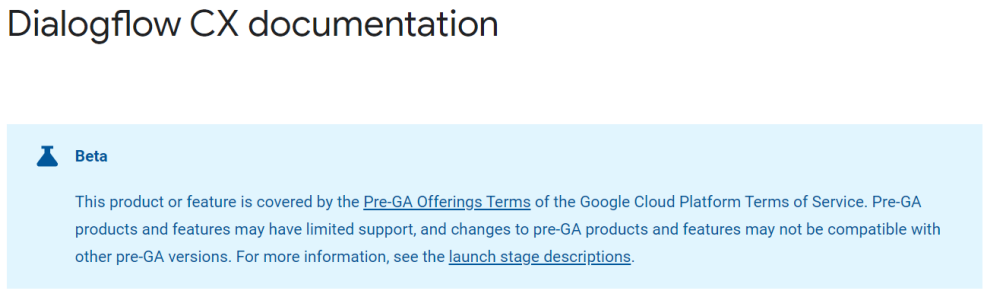


Figura 6.1: Warning presente nella documentazione di Dialogflow CX quando è stata esaminata in fase di progettazione. Fonte: [40]

All'atto pratico questo avrebbe potuto comportare un cambio improvviso delle funzionalità o delle API offerte da Google, richiedendo un adeguamento dell'applicativo oggetto di questa tesi. Inoltre, altro aspetto problematico, Dialogflow CX non supportava inizialmente la creazione di *chatbot* in italiano.

Riguardo Dialogflow ES, dopo un confronto con l'azienda, si è convenuto che fosse troppo a rischio di essere rimpiazzata in un futuro non poi tanto remoto dalla versione CX, o che per lo meno fosse a rischio di essere accantonata da Google.

Queste considerazioni hanno portato a scartare Dialogflow come piattaforma generalista di appoggio.

IBM Watson Un'altra delle alternative esaminate è stata IBM Watson Assistant. A differenza di Dialogflow, la piattaforma veniva (e viene tutt'ora) offerta in un'unica versione e con il supporto alla lingua italiana.

Una potenziale criticità individuata risiede però nelle API offerte. Infatti IBM sta offrendo due versioni delle API per l'interazione con l'assistente: *v1* e *v2*. Il problema è che quelle più recenti, ovvero le *v2*, non supportano la creazione, modifica e cancellazione di *chatbot*. D'altro canto le API *v1*, in accordo con l'azienda, sono state valutate troppo a rischio di essere rimpiazzate. Proprio questa instabilità delle API ha portato a non affidarsi a Watson Assistant come piattaforma generalista di appoggio.

Amazon Lex L'ultima alternativa analizzata è stata Amazon Lex. Le peculiarità riscontrate della piattaforma sono state:

- Il supporto alla lingua italiana, introdotto a metà novembre 2020;
- L'utilizzo dello stesso motore di NLP di *Alexa*, che è la soluzione di smart speaker più diffusa sul mercato [39];
- La possibilità di importare ed esportare *chatbot*, intenti ed entità via file *.json*, rendendo di fatto implementabile in maniera diretta la funzionalità di esportazione di un *chatbot* creato con l'applicazione oggetto di questa tesi verso Lex;
- La possibilità di trasformare un *chatbot* in skill per *Alexa*, fatto che apre le porte ad innumerevoli scenari, soprattutto in ottica aziendale;

Queste caratteristiche, unite alle criticità riscontrate nei competitor, hanno portato alla scelta di Amazon Lex come piattaforma generalista di supporto.

6.2 Stack tecnologico

La scelta delle tecnologie utilizzate nello sviluppo di questa applicazione è stata fortemente influenzata dalle preferenze espresse dall'azienda e dal suo *know-how*. Primo Round infatti ha posto come requisito che venisse realizzata una SPA e, considerando la sua grande esperienza nell'utilizzo dello stack MEAN, la scelta è ricaduta su queste tecnologie, le cui caratteristiche vengono approfondite nei paragrafi successivi.

6.2.1 Frontend: Angular

Angular [21] è un framework open-source per la realizzazione di SPA sviluppato da Google a partire dal 2016. Le applicazioni vengono scritte utilizzando *TypeScript* [42], un'estensione open source del linguaggio *JavaScript* sviluppata da Microsoft. Le caratteristiche principali di *TypeScript* sono:

- Tipizzazione statica delle variabili: in TypeScript ogni variabile deve avere un tipo definito a priori. Questo consente al compilatore di validare in modo più efficace il codice, evidenziando prima dell'esecuzione eventuali comportamenti inattesi;
- Compilazione del codice in JavaScript, rendendolo così compatibile con qualsiasi dispositivo capace di eseguire codice JavaScript;

- Aderenza allo standard *EcmaScript 6* [26] che include, tra le altre cose:
  - Supporto a classi, interfacce ed ereditarietà;
  - Supporto alle "arrow function" che consentono di scrivere espressioni *Lambda*;
  - Supporto integrato alla gestione (importazione/esportazione) di moduli, che consente di compartimentare meglio il codice realizzato, rendendo più facile l'adesione al SRP.

Un'applicazione Angular è formata da *componenti*, *servizi* e *moduli*.

Componenti I componenti sono degli elementi che incapsulano una funzionalità grafica dell'applicazione. Un componente può essere una pagina, una finestra modale, un bottone o persino una particolare porzione di testo. Ogni componente è formato da tre elementi:

- La logica, definita in un file *TypeScript* del tipo `NomeComponente.ts`;
- Il template scritto in HTML che verrà usato per il rendering del componente. Il template può essere definito internamente (inline) al file *NomeComponente.ts* oppure trovarsi in un file *.html* separato, convenzionalmente denominato `NomeComponente.html`;
- Il foglio di stile CSS, LESS o *sass* che definisce la grafica del componente. Il foglio di stile può risiedere internamente al file TypeScript, oppure essere posizionato in un file separato. Angular inoltre di default usa un approccio chiamato *view encapsulation*: lo stile personalizzato definito in un componente non va a "contaminare" quello degli altri componenti dell'applicazione.

Servizi I servizi sono dei particolari tipi di classi che implementano una specifica porzione della logica di business dell'applicazione. Idealmente un componente dovrebbe occuparsi solo di visualizzare informazioni e raccogliere l'input dell'utente, mentre tutti i task più complessi, come chiamate ad API esterne o validazione dell'input, dovrebbero essere gestite dai servizi.

Per fornire un servizio ad un componente che ne fa richiesta, Angular fa uso del design pattern Dependency Injection. In pratica, quando un componente viene creato, Angular determina quali sono i servizi che esso richiede e si occupa di istanziarli. Delegando l'istanziazione del servizio all'esterno del componente, se più componenti richiedono lo stesso servizio Angular passerà loro la stessa istanza. Questo approccio presenta due vantaggi: la comunicazione tra componenti può essere astratta nel servizio e l'utilizzo della memoria viene ottimizzato.

Moduli In Angular i moduli sono i "mattoncini" che compongono l'applicazione, contenenti porzioni di codice inerente ad una certa funzionalità. Al suo interno, un modulo può contenere componenti, servizi o altri moduli. Tutto questo viene definito in un file *TypeScript* dedicato, del tipo `nomemodulo.module.ts`.

Ogni applicazione ha almeno un modulo, chiamato *AppModule*, che è quello utilizzato da Angular per il boostrap della pagina web. Facendo un paragone con i linguaggi di programmazione *Java* o *C++*, *AppModule* equivale alla classe che contiene il metodo *main*.

6.2.2 Backend: NestJS

Per l'implementazione del backend la scelta è ricaduta su NestJS [37], un framework basato su *TypeScript* per la costruzione di applicazioni web con Node.js e Express.js.

Il vantaggio di utilizzare NestJS rispetto ad altri framework è la somiglianza con Angular. Infatti, oltre a condividere l'utilizzo di *TypeScript*, entrambi i framework utilizzano i moduli

per compartimentare il codice, i servizi per gestire la logica di business e il design pattern Dependency Injection.

Per gestire la creazione di API in maniera semplice, NestJS fa uso dei *controller*, cioè particolari tipi di classi responsabili dell'interazione con il client. A livello pratico, un controller è una classe TypeScript a cui vengono applicate dei particolari decorator. Nello specifico:

- A livello di classe, viene utilizzato `@Controller('/url/gestito/')` che definisce che il controller rappresentato dalla classe decorata intercetterà le chiamate che arrivano all'url "/url/gestito";
- A livello di metodo sono disponibili le notazioni `@Get`, `@Post`, `@Put`, `@Delete` per far si che il metodo annotato gestisca la corrispondente richiesta HTTP.

6.2.3 Database: MongoDB

MongoDB [36] è un DBMS non relazionale. Ogni record presente nel database è chiamato *documento* e assume una forma simile ad un JSON. I documenti sono raggruppati in *collection*. La caratteristica di questo approccio, differenza sostanziale rispetto ai DBMS relazionali, è che i documenti non assumono una struttura definita a priori: all'interno di una collection si possono salvare documenti aventi attributi diversi, rendendo così la base di dati più flessibile.

6.3 Architettura della soluzione

Una volta definite le tecnologie è stato il momento di progettare l'architettura dell'applicazione, ovvero di definire i suoi componenti e le modalità di interazione tra essi, con un focus particolare sullo schema di comunicazione tra questa piattaforma e Amazon Lex.

Come spiegato nella sezione 6.2, l'applicazione è costruita su stack MEAN, quindi è composta da tre elementi: *frontend*, *backend* e DBMS. L'interazione tra frontend e backend avviene tramite delle richieste HTTP asincrone (AJAX), mentre quella tra backend e DBMS sfrutta il driver *mongoose*. Inoltre si è resa necessaria la creazione di una funzione AWS Lambda per la comunicazione tra Lex e l'applicazione: ulteriori dettagli sulle motivazioni alla base di questa decisione verranno esposte in sezione 6.3.3. Una rappresentazione grafica ad alto livello dell'applicazione è fornita in figura 6.2.

6.3.1 Struttura del backend

Il backend adotta una struttura di tipo modulare, ovvero ogni funzionalità è stata incapsulata in un modulo a sé stante. I moduli presenti sono: *Users*, *Auth*, *Database*, *Chatbots*, *Lex*, *Answers* e *FileImport*.

Users Lo scopo di questo modulo è la gestione di tutto ciò che riguarda gli utenti dell'applicazione. Verranno quindi implementate ed esposte delle funzionalità che permetteranno di salvare un nuovo utente nella base di dati, modificare o eliminare uno esistente e ottenere la lista degli utenti presenti.

Auth Questo modulo implementa tutte le funzionalità inerenti all'autenticazione di un utente: login, registrazione, e gestione dei token *JWT*.

Database Incapsula le informazioni e le configurazioni necessarie all'interazione con il database.

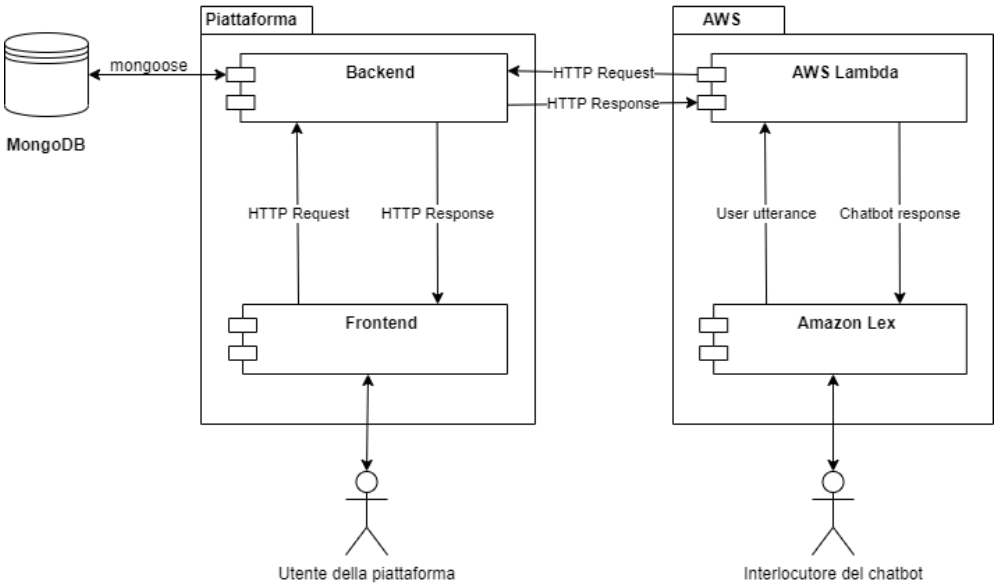


Figura 6.2: Visione ad alto livello dell'architettura dell'applicazione

Chatbots Contiene tutte le funzionalità necessarie alla creazione e alla gestione di un *chatbot*. A sua volta contiene i moduli:

- `Knowledges` per la gestione della base di conoscenza;
- `Themes` per la gestione dei temi;
- `Needs` per la gestione dei bisogni;
- `Questions` per la gestione della lista delle domande di un bisogno, cioè delle *utterance* trigger.

Answers Ingloba tutte le funzionalità necessarie alla creazione, modifica, eliminazione delle risposte. Come nel caso di *Chatbots*, è composto da sotto-moduli:

- `SimpleTextAnswers` che gestisce le risposte di tipo "Testo semplice",
- `ImageAnswers` si occupa delle risposte multimediali formate da un'unica immagine;
- `VideoAnswers` è l'omologo di *ImageAnswer* nel caso in cui la risposta sia un video;
- `OptionAnswers` gestisce le opzioni di risposta (quindi le scelte) nel caso di risposta di tipo "Dipende Da".

FileImport Si pone di gestire il caricamento, il parsing e il salvataggio del file in formato Excel contentente una *Knowledge Base* esistente.

Lex Incapsula tutte le funzionalità che riguardano l'interazione di qualsiasi tipo con Amazon Lex. A sua volta è costituito da:

- `LexJSON` che fornisce i metodi necessari all'esportazione del file JSON del *chatbot*, oltre che all'importazione nell'applicazione di un *chatbot* esportato da Lex;
- `LexLambda` che implementa l'interazione con la funzione AWS Lambda per la gestione della logica del *chatbot*, che verrà descritta nel dettaglio in sezione 6.3.3;
- `LexInstaller`, un modulo che permette di installare in maniera automatica la funzione AWS Lambda in un account AWS esistente, configurandone anche i permessi per l'esecuzione.

6.3.2 Struttura del frontend

Il frontend adotta una struttura leggermente diversa: invece di incapsulare le funzionalità in più moduli, è si è optato per un approccio a componenti. Questo significa che si è progettato un componente:

- Per ogni schermata dell'applicazione;
- Per la gestione di ogni funzionalità (grafica) richiesta molteplici volte nell'applicazione.

I componenti principali sono **Chatbots**, **Wizard** e **Login**, il cui scopo e struttura vengono ora accennati.

Chatbots Si occupa di visualizzare la pagina iniziale con la lista di tutti i *chatbot* esistenti e il pulsante che fa partire il wizard di creazione di un nuovo bot. In pratica è incaricato di visualizzare la schermata in figura 5.1. Per il rendering di ogni elemento della lista dei *chatbot* già inseriti verrà utilizzato il componente **ChatbotListItem**;

Wizard Viene evocato quando l'utente decide di creare un nuovo *chatbot*. Funge da "contenitore" per tutti gli step del tutorial. Sarà suo compito la gestione della guida contestuale e della barra di avanzamento. In ogni caso farà uso di un sotto-componente per ogni step del tutorial da visualizzare:

- `ChatbotIdentity` gestirà la schermata con l'inserimento dei dettagli dell'identità del *chatbot* (figura 5.2);
- `FirstQuestion` si occuperà di far inserire all'utente la prima domanda del bisogno, oltre che a fornire il form per l'upload del file contentente una *knowledge base* esistente (5.3);
- `QuestionsList` permetterà all'utente di gestire la lista delle domande di un bisogno, coerentemente con quanto mostrato in figura 5.4;
- `Need` si occuperà di far inserire (ed eventualmente modificare) all'utente il nome del bisogno corrente (5.5);
- `AnswerTypePicker` permetterà la selezione del tipo di risposta al bisogno, unitamente all'inserimento del tipo di risposta "semplice" testuale o multimediale (immagine o video), come mostrato in figura 5.7;
- `DependOn` gestirà il tipo di risposta "Dipende da" (figura 5.8);
- `Theme` si occuperà di far inserire (ed eventualmente modificare) all'utente il nome del tema corrente (5.6);

- `Summary` visualizzerà la schermata di riepilogo al termine dell'addestramento, come mostrato in figura 5.11

Infine sarà presente un altro componente, chiamato **KnowledgeRepresentation**, dedicato alla rappresentazione insiemistica della base di conoscenza presente nella maggior parte degli step del tutorial di addestramento.

Login che gestisce la schermata di login e registrazione, non presente nel mockup fornito dall'azienda ma comunque necessaria;

6.3.3 Interazione con Amazon Lex

Fino a questo momento non è mai stata menzionata la modalità con la quale l'applicazione dovrebbe interagire con Amazon Lex. A tal proposito, in fase di progettazione si sono dovuti affrontare due problemi principali, nell'ordine in cui vengono menzionati:

1. Come poter esportare il *chatbot* creato con l'applicazione ed importarlo in Amazon Lex;
2. Come gestire la risposta di tipo "dipende da" in un bisogno;

Ciascuno di questi aspetti viene ora analizzato individualmente.

6.3.3.1 Importazione del *chatbot* in Amazon Lex

Per importare un *chatbot* in Amazon Lex esistono due soluzioni percorribili:

- L'utilizzo della console di gestione di Lex per l'upload manuale del *chatbot*;
- L'utilizzo delle API proprietarie per un deploy automatizzato.

In entrambi i casi il *chatbot* deve essere descritto tramite un particolare file JSON che contiene il nome, la lista degli intenti e delle entità, la lingua ed altri parametri di configurazione.

Visto che la generazione del file JSON è un'operazione richiesta da entrambe le soluzioni, si è optato per la prima soluzione: creare il file con la descrizione del *chatbot*, farlo scaricare all'utente e caricarlo manualmente in Lex. Se poi in un futuro si volesse automatizzare maggiormente il processo di deploy, sarà necessario solamente implementare le API fornite da Amazon per caricare il JSON realizzato dall'applicazione.

Il problema successivo affrontato è stato tradurre la struttura di *chatbot*, temi e bisogni utilizzata in un formato comprensibile a Lex. Analizzando quest'ultimo formato si nota come il file JSON da generare descrive un *chatbot* e contiene i seguenti attributi significativi:

- `name`: il nome del bot. Si è deciso di mappare questo attributo con il nome del *chatbot* che l'utente inserisce all'inizio del wizard;
- `locale`: la lingua adottata. Siccome l'applicazione realizzata è in italiano, il valore è stato impostato staticamente a `it-IT`, in modo che si rescano a gestire conversazioni in lingua italiana;
- `intents`: la lista degli intenti che il bot gestisce. Per la metafora del "bisogno" utilizzata nell'applicazione, è sembrato naturale mappare ogni bisogno ad un intento;
- `slotTypes`: la lista delle entità personalizzate che il *chatbot* utilizza nei suoi intenti. Le entità si sono rivelate particolarmente utili nella gestione del tipo di risposta "Dipende Da" (maggiori dettagli in sezione 6.3.3.2);

- `clarificationPrompt`: la frase che verrà utilizzata dal *chatbot* per comunicare all'utente che non è riuscito a riconoscere la usa richiesta. Si è deciso di impostare questa opzione "staticamente" a "Non ho capito, puoi provare a riformulare la domanda?".
- `abortStatement`: la frase che viene utilizzata quando il *chatbot* non è riuscito a riconoscere l'input dell'utente molteplici volte di seguito, quindi decide di terminare la conversazione: è stato impostato staticamente a "Mi dispiace, ma non riesco ad aiutarti.".

Ricapitolando un punto fondamentale, si è deciso di creare, per ogni bisogno presente nel *chatbot*, un intento, che dovrà essere anch'esso descritto con una particolare struttura JSON. Gli attributi più significativi del formato JSON richiesto da Lex per la descrizione di un intento sono:

- `name`: il nome dell'intento. Siccome deve essere univoco non solo nel contesto del *chatbot*, ma in tutto l'account AWS in cui il *chatbot* risiede, si è deciso di mapparlo ad un particolare codice formato da `nome dell'intento + id del chatbot`;
- `sampleUtterances`: la lista delle *utterance* associate all'intento. Questo campo viene riempito con la lista delle domande del bisogno, che concretamente sono quelle inserite nella schermata 5.4;
- `conclusionStatement`: la risposta che il *chatbot* deve ritornare all'utente una volta "completato" l'intento. Questo campo viene valorizzato con il contenuto della risposta del bisogno nel caso questa sia "diretta", quindi non del tipo "Dipende Da". In caso contrario il campo viene inizializzato al valore di default "Spero di esserti stato utile, buona giornata!";
- `slots`, `slotTypes`, `fulfillmentActivity` e `dialogCodeHook` sono dei campi che vengono utilizzati per la gestione della risposta di tipo "Dipende Da" e la loro utilità verrà approfondita in sezione 6.3.3.2;

6.3.3.2 Gestione della risposta di tipo "Dipende Da"

Il limite maggiore riscontrato già in fase di analisi della piattaforma Amazon Lex è l'impossibilità di gestire istruzioni condizionali in modo diretto all'interno di un intento. Per implementare interazioni più complesse, l'approccio utilizzato da Lex è quello di delegare il calcolo della risposta ad una funzione AWS Lambda . Quest'ultima è consiste in un codice, scritto sotto forma di funzione, che viene eseguito su un'infrastruttura *serverless* (*AWS Lambda*), ovvero senza necessità di doversi preoccupare dell'allocazione delle risorse necessarie. Ad esempio, se nella funzione Lambda viene scritto del codice javascript, sarà AWS che si occuperà, al momento dell'esecuzione, dell'allocazione di una macchina virtuale con la corretta versione di *Node.js* e un sufficiente spazio di memoria disponibile.

Ritornando alla gestione della risposta "Dipende Da", la funzione AWS Lambda riceverà in input delle informazioni relative al dialogo in corso (tra cui l'*utterance* dell'utente e l'intento corrente), mentre come output fornirà il "comportamento" a cui il bot dovrà attenersi per il proseguo del dialogo. Al bot può essere ordinato di richiedere un'informazione o un chiarimento all'utente oppure restituire a quest'ultimo una risposta, chiudendo così il dialogo.

Lex può invocare la funzione AWS Lambda in due momenti distinti:

- In seguito ad *ogni* interazione dell'utente. In questo caso il riferimento alla funzione da invocare va inserito nel campo `dialogCodeHook` del JSON che descrive l'intento;
- In fase di *fulfillment*, ovvero quando l'intento è stato riconosciuto e tutti i suoi *slot*, se presenti, sono stati valorizzati correttamente. In questo caso bisogna inserire il riferimento alla funzione nel campo `fulfillmentActivity` del JSON dell'intento.

6.3.3.2.1 Parametri della funzione Lambda Le informazioni passate in input alla funzione (sotto forma di JSON) includono:

- `currentIntent`, cioè l' "oggetto" attuale del dialogo. Viene descritto da numerosi parametri, ma quelli più di interesse sono:
  - `name`, cioè il nome dell'intento;
  - `slots`, una mappa che associa ad ogni *slot* (identificato tramite il suo nome) il valore a cui è stato inizializzato, se presente;
- `sessionAttributes`, una mappa che permette di salvare degli attributi personalizzati. Questi vengono utilizzati esclusivamente dalla funzione AWS Lambda e consentono più facilmente di tener traccia dello stato della conversazione. Il loro valore viene completamente ignorato da Lex, che si occupa solo di memorizzarli e fornirli alla Lambda ad ogni invocazione.

La risposta ritornata dalla funzione è anch'essa in formato JSON e include i seguenti attributi:

- `sessionAttributes`, una versione "aggiornata" dell'omonima mappa ricevuta come input, che Lex allegherà come input alla successiva richiesta;
- `dialogAction`, che serve a fornire a Lex le istruzioni su come comportarsi, ovvero se e cosa restituire all'utente e cosa aspettarsi con l'interazione successiva. Il valore di questo attributo è di tipo JSON, tra i cui campi c'è l'attributo `type`. Esso determina il comportamento successivo del bot; determina anche gli altri campi che la funzione Lambda deve fornire come parte del valore `dialogAction`. Può essere:
  - `Close`: il bot non si aspetterà più una risposta dall'utente poiché l'interazione con esso è da considerarsi terminata. In questo caso è possibile allegare un messaggio da ritornare all'utente tramite il campo `message`. Ad esempio, nel caso di un intento che restituisca le previsioni meteo, il JSON di risposta potrebbe la struttura del listato 6.1;

```
"sessionAttributes": {
    "localita": "Padova",
    "data": "2021-04-22",
    ....
}
"dialogAction": {
    "type": "Close",
    "message": {
        "contentType": "PlainText",
        "Content": "Le previsioni per Padova
   indicano cielo sereno il giorno 22 Aprile."
    }
}
```

Listing 6.1: Esempio di risposta con dialogAction di tipo Close

  - `ElicitSlot`: il bot dovrà chiedere all'utente di valorizzare un'informazione contenuta in uno *slot*. In questo caso è necessario specificare tre parametri:
    * `intentName`: l'intento che contiene lo *slot* da valorizzare;

  - `slots`, cioé una mappa che associa ad ogni *slot* dell'intento il valore corrente;
  - `slotToElicit`: il nome dello *slot* da valorizzare.

Riprendendo l'esempio del meteo, poniamo il caso che si voglia richiedere all'utente la località. Il JSON allora potrà assumere la struttura del listato 6.2.

```
"sessionAttributes": {
    "data": "2021-04-22",
    ....
}
"dialogAction": {
    "type": "ElicitSlot",
    "intentName": "infoMeteo",
    "slots": {
        "data": "2021-04-22",
        "localita": null
    },
    "slotToElicit": "localita",
}
```

Listing 6.2: Esempio di risposta con dialogAction di tipo ElicitSlot

– `ElicitIntent`: viene usato quando il bot non è riuscito a riconoscere completamente l'intento e deve chiedere un chiarimento all'utente. Opzionalmente è possibile specificare la domanda che il bot porrà all'utente tramite il parametro `message`; in caso non venga specificato Lex userà il valore di *clarificationPrompt* presente nel JSON del *chatbot*. Ad esempio, poniamo che l'espressione dell'utente non sia stata riconosciuta dal *chatbot*. Questi potrà richiedere un chiarimento all'utente con il JSON del listato 6.3.

```
"sessionAttributes": {
    "chiaveUno": "valoreUno",
    "chiaveDue": "valoreDue",
    ....
}
"dialogAction": {
    "type": "ElicitIntent",
    "message": {
        "contentType": "PlainText",
        "Content": "Non ho capito. Potresti
   ripetere la domanda?"
    }
}
```

Listing 6.3: Esempio di risposta con dialogAction di tipo ElicitIntent

– `Delegate`, che lascia decidere a Lex l'azione successiva da compiere.

Per un elenco completo di tutti i parametri presenti, nonché di tutte le opzioni possibili, si rimanda a [29].

6.3.3.2.2 Modalità di utilizzo della funzione Lambda In un intento con tipo di risposta "Dipende da" viene creato uno *slot* per ogni nodo decisionale. Il tipo dello *slot* sarà un'entità (*SlotType*) personalizzata, i cui valori corrisponderanno alle possibili opzioni di risposta. Durante il dialogo con l'utente, il flusso conversazionale viene percorso dalla radice fino alle foglie, tenendo traccia dei nodi visitati grazie ai `sessionAttributes`. Quando il nodo corrente sarà di tipo decisionale, la funzione AWS Lambda restituirà una *dialogAction* di tipo `ElicitSlot`, avente come `slotName` il riferimento allo *slot* corrispondente al nodo decisionale corrente. Questo farà in modo che Lex porrà all'utente la domanda relativa al nodo corrente, salvando la sua scelta in uno *slot*. Una volta giunti ad una foglia, verrà emessa una `dialogAction` di tipo `Close` che permetterà di restituire la risposta finale dell'intento all'utente. Vista la complessità della soluzione, verrà ora fornito un esempio pratico del funzionamento.

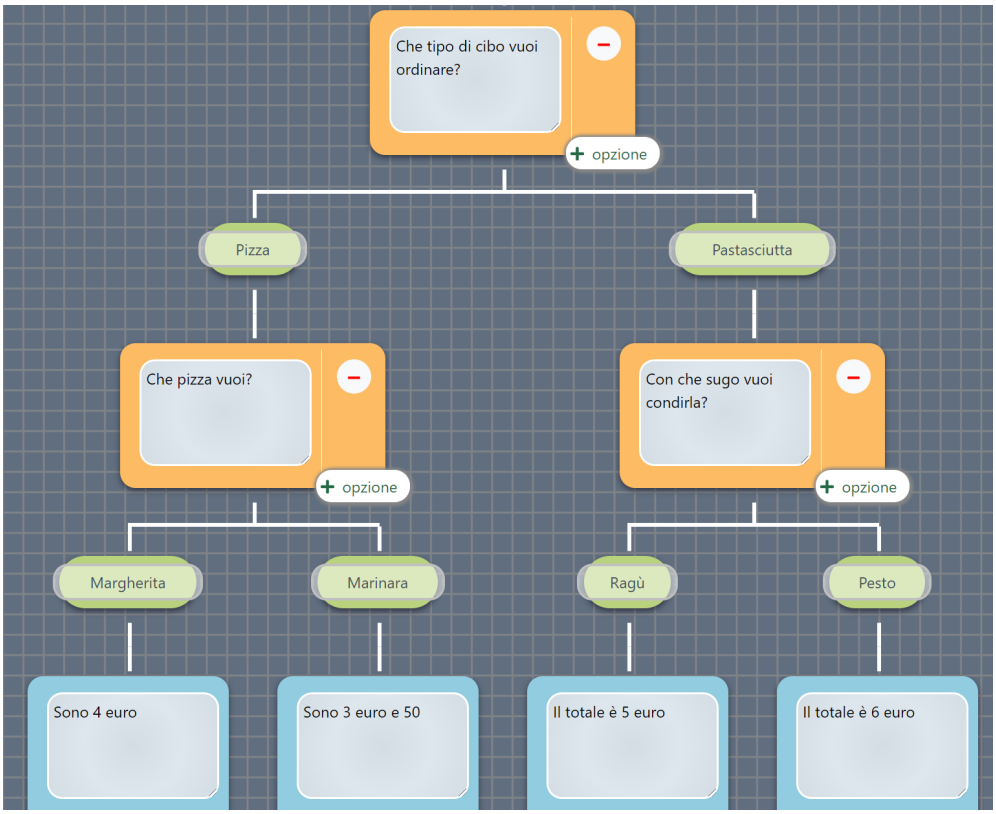


Figura 6.3: Diagramma di una risposta di tipo "Dipende Da" realizzato con l'applicativo oggetto della tesi

Supponiamo di avere un bisogno, chiamato "NuovoOrdine", che permette ad un utente di ordinare del cibo con il *chatbot*. Questo bisogno adotterà una risposta di tipo "Dipende Da", il cui diagramma di flusso è riportato in 6.3. Nell'immagine si possono individuare tre tipi di nodi, in base al loro colore di sfondo:

- Nodi *decisionali*, con sfondo arancione, che pongono una domanda all'utente;
- Nodi *opzione*, con sfondo verde, che indicano un'opzione di risposta valida per il nodo decisionale padre;

- Nodi *risposta*, con sfondo azzurro, che contengono la risposta finale da restituire all'utente. Questi sono anche nodi foglia, in quanto non ammettono figli.

Ipotizziamo che ad ogni nodo sia assegnato un codice univoco formato dalla lettera (in maiuscolo) dell'alfabeto corrispondente alla posizione del nodo nell'ordine topologico di esplorazione dell'albero. Ovvero:

- Il nodo con domanda "Che tipo di cibo vuoi ordinare?" avrà come codice `A`;
- Il nodo con l'opzione "Pizza" avrà come codice `B`;
- Il nodo con l'opzione "Pastasciutta" avrà come codice `C`;
- Il nodo con domanda "Che pizza vuoi?" avrà come codice `D`;
- Il nodo con domanda "Con che sugo vuoi condirla?" avrà come codice `E`;
- Il nodo con l'opzione "Margherita" avrà come codice `F`;
- Il nodo con l'opzione "Marinara" avrà come codice `G`;
- Il nodo con l'opzione "Ragù" avrà come codice `H`;
- Il nodo con l'opzione "Pesto" avrà come codice `I`;
- Il nodo con risposta "Sono 4 euro" avrà come codice `J`;
- Il nodo con risposta "Sono 3 euro e 50" avrà come codice `K`;
- Il nodo con risposta "Il totale è 5 euro" avrà come codice `L`;
- Il nodo con risposta "Il totale è 6 euro" avrà come codice `M`;

Siccome i nodi decisionali sono tre, l'intento associato al bisogno "NuovoOrdine" avrà tre *slot*:

- `pietanza`, che conterrà la risposta dell'utente alla domanda "Che tipo di cibo vuoi ordinare?";
- `tipoDiPizza`, che conterrà la risposta dell'utente alla domanda "Che pizza vuoi?";
- `sugoDellaPasta` che conterrà la risposta dell'utente alla domanda "Con che sughi vuoi condirla?".

Al fine di limitare le opzioni di risposta ammesse, per ogni *slot* verrà creata un'entità (*SlotType*) personalizzata. Ovvero le seguenti entità verranno create:

- `PietanzaEntity`, i cui valori saranno "Pizza" e "Pastasciutta";
- `TipoDiPizzaEntity`, i cui valori saranno "Margherita" e "Marinara";
- `SugoDellaPastaEntity`, i cui valori saranno "Ragù" e "Pesto.

Siccome le entità personalizzate devono essere inserite nell'attributo `slotTypes` del JSON del *chatbot*, quest'ultimo assumerà la struttura del listato 6.4.

```
"slotTypes": [{
  "name": "PietanzaEntity",
  "enumerationValues": [{
    "value": "Pizza"
  }, {
   "value": "Pastasciutta"
  }],
},
{
  "name": "TipoDiPizzaEntity",
  "enumerationValues": [{
    "value": "Margherita"
  },{
    "value": "Marinara"
  }],
}, {
  "name": "SugoDellaPastaEntity",
  "enumerationValues": [{
    "value": "Ragu"
  },{
    "value": "Pesto"
  }],
}],
```

Listing 6.4: Valore dell'attributo `slotTypes` del JSON del *chatbot* che contiene l'intento "NuovoOrdine"

L'attributo `slots` del JSON relativo all'intento "NuovoOrdine", invece, assumerà la struttura del listato 6.5.

```
"slots": [{
  "name": "pietanza",
  "slotType": "PietanzaEntity",
  "slotConstraint": "Optional",
  "valueElicitationPrompt": {
    "messages": [{
      "contentType": "PlainText",
      "content": "Che tipo di cibo vuoi ordinare?"
    }],
  },
  "priority": 1
}, {
  "name": "tipoDiPizza"
  "slotType": "TipoDiPizzaEntity",
  "slotConstraint": "Optional",
  "valueElicitationPrompt": {
    "messages": [{
      "contentType": "PlainText",
      "content": "Che pizza vuoi?"
    }],
  },
  "priority": 2
```

```
}, {
  "name": "sugoDellaPasta"
  "slotType": "SugoDellaPastaEntity",
  "slotConstraint": "Optional",
  "valueElicitationPrompt": {
    "messages": [{
      "contentType": "PlainText",
      "content": "Con che sugo vuoi condirla?"
    }],
  },
  "priority": 3
}]
```

Listing 6.5: Valore dell'attributo `slots` dell'intento "NuovoOrdine"

Ora, supponiamo che il *chatbot* venga importato in Lex e di essere nel mezzo di una conversazione con l'utente. Ipotizziamo che il *chatbot* abbia appena riconosciuto l'intento "NuovoOrdine". Lex farà una chiamata alla funzione AWS Lambda con i parametri mostrati nel listato 6.6.

```
{
  "currentIntent": {
    "name": "NuovoOrdine",
    "slots": {
      "pietanza": null,
      "tipoDiPizza": null,
      "sugoDellaPasta": null
    }
  },
  "sessionAttributes": {},
}
```

Listing 6.6: Richiesta inviata alla funzione AWS Lambda alla prima interazione

A questo punto la funzione AWS Lambda dovrà calcolare la risposta da fornire a Lex. Per farlo esaminerà i parametri `currentIntent`, `slots` e `sessionAttributes` della richiesta. In particolare:

- `currentIntent` non è nullo. Questo significa che Lex è riuscito a riconoscere correttamente l'intenzione dell'utente;
- `sessionAttributes` non contiene nessuna informazione, quindi si può dedurre che non è avvenuta alcuna interazione con l'utente, al di fuori della *utterance* di quest'ultimo che ha portato all'identificazione dell'intento;
- Tutti gli `slots` hanno valore *null*. Questo significa che sicuramente il *chatbot* dovrà chiederne la valorizzazione.

La funzione AWS Lambda ora inizierà con lo scorrere il flusso conversazionale della risposta associata all'intento, rappresentato graficamente nell'immagine 6.3. Il primo nodo è di tipo *decisionale* ed è stato associato, in fase di creazione del JSON, allo slot `pietanza` di tipo `PietanzaEntity`. Questo implica due cose:

- La funzione dovrà quindi "ordinare" al *chatbot* di chiedere all'utente che tipo di pietanza voglia ordinare. In pratica dovrà comunicare a Lex la necessità di valorizzare lo slot di nome "pietanza";
- Avendo visitato un nodo "nuovo", in `sessionAttributes` dovrà essere inserito un riferimento a quest'ultimo, che in questo caso è rappresentato dalla lettera "A".

Ne consegue che la risposta ritornata a Lex avrà il contenuto del listato 6.7.

```
{
  "dialogAction": {
    "type": "ElicitSlot",
    "intentName": "NuovoOrdine",
    "slots": {
      "pietanza": null,
      "tipoDiPizza": null,
      "sugoDellaPasta": null
    },
    "slotToElicit": "pietanza"
  },
  "sessionAttributes": {
    "0": "A"
  }
}
```

Listing 6.7: Risposta della funzione AWS Lambda alla prima interazione

Una piccola precisazione sulla struttura dei dati inseriti in `sessionAttributes`: siccome è una mappa, memorizza coppie di tipo *chiave-valore*. Si è deciso di utilizzare un numero come chiave, che corrisponde all'ordine di esplorazione del nodo. In questo caso `"0":"A"` indica che il nodo "A" è stato esplorato per primo (la numerazione parte da 0).

Il listato 6.7 farà sì che il *chatbot* porrà all'utente il quesito "Che tipo di cibo vuoi ordinare?", presentando all'utente due opzioni di risposta, ovvero "Pizza" e "Pasta". Supponiamo che l'utente abbia selezionato "Pizza": Lex farà una chiamata alla funzione AWS Lambda passando i parametri mostrati dal listato 6.8.

```
{
  "currentIntent": {
    "name": "NuovoOrdine",
    "slots": {
      "pietanza": "Pizza",
      "tipoDiPizza": null,
      "sugoDellaPasta": null
    }
  },
  "sessionAttributes": {
    "0": "A"
  }
}
```

Listing 6.8: Richiesta inviata alla funzione AWS Lambda alla seconda interazione

Come prima cosa la funzione AWS Lambda esaminerà il parametro `sessionAttributes`, notando che non è vuoto. Ordinerà allora i suoi valori per chiave decrescente e prenderà il

valore del primo di essi, in modo da recuperare lo stato del dialogo precedente all'interazione appena avvenuta con l'utente, ovvero in questo caso il riferimento al nodo "A". Consultando il flusso di conversazione noterà che "A" è il nodo *decisionale* corrispondente allo slot "pietanza", che risulta valorizzato a "Pizza". Questo significa che l'ultimo nodo del flusso conversazionale visitato è in realtà "B", che quindi verrà inserito in `sessionAttributes` con chiave "1". Una volta ricostruito lo stato del dialogo, ovvero l'ultima interazione avvenuta con l'utente, la funzione AWS Lambda passerà al calcolo della risposta. A tal proposito, noterà come il nodo figlio di "B" sia un nodo *decisionale* corrispondente allo slot `tipoPizza`, che non risulta valorizzato. Pertanto, analogamente a quanto avvenuto con l'interazione precedente, inserirà il valore `"2":"D"` nella mappa `sessionAttributes` ed emetterà una `dialogAction` di tipo `ElicitSlot` con `slotName` impostato a `tipoPizza`. L'output completo viene riportato per chiarezza nel listato 6.9.

```
{
  "dialogAction": {
    "type": "ElicitSlot",
    "intentName": "NuovoOrdine",
    "slots": {
      "pietanza": "Pizza",
      "tipoDiPizza": null,
      "sugoDellaPasta": null
    },
    "slotToElicit": "pizza"
  },
  "sessionAttributes": {
    "2": "D",
    "1": "B",
    "0": "A"
  }
}
```

Listing 6.9: Risposta della funzione AWS Lambda alla seconda interazione

Lex di conseguenza chiederà all'utente di riempire lo slot `tipoPizza` ponendoli la domanda "Che pizza vuoi ordinare?" e presentandogli le opzioni di risposta "Margherita" e "Marinara". Supponiamo che l'utente scelga "Marinara": Lex comunicherà questa informazione alla funzione AWS Lambda tramite una chiamata con l'input del listato 6.10.

```
{
  "currentIntent": {
    "name": "NuovoOrdine",
    "slots": {
      "pietanza": "Pizza",
      "tipoDiPizza": "Marinara",
      "sugoDellaPasta": null
    }
  },
  "sessionAttributes": {
    "2": "D",
    "1": "B",
    "0": "A"
  }
```

```
15 }
```

Listing 6.10: Richiesta inviata alla funzione AWS Lambda alla terza interazione

Analogamente a prima la funzione AWS Lambda ordinerà il contenuto del parametro `sessionAttributes` per chiave decrescente, prendendo il primo valore, quindi "D". Noterà inoltre che esso fa riferimento ad un nodo *decisionale* e che lo slot associato, quindi `tipoPizza`, risulta valorizzato a "Marinara". Inserirà allora in `sessionAttributes` il riferimento al nodo "G" (con chiave "3", quindi verrà inserita la coppia `"3":"G"`) poiché corrisponde all'opzione selezionata dall'utente. In questo modo avrà ricostruito il fatto che l'ultima interazione con l'utente ha portato il dialogo in corrispondenza del nodo "G". Proseguendo con il calcolo della risposta, noterà che "G" ha un unico figlio di tipo *risposta*, avente codice "K". Verrà allora aggiorato il parametro `sessionAttributes` inserendo la coppia `"4":"K"` e, siccome "K" è un nodo foglia, verrà chiesto di concludere il dialogo impostando il parametro `dialogAction` a `Close` e l'attributo `Content` del parametro `message` a "Sono 3 euro e 50", ovvero la risposta che verrà restituita all'utente. L'output della funzione è riassunto nel listato 6.11.

```
1  {
2    "dialogAction": {
3      "type": "Close",
4      "message": {
5        "contentType": "PlainText",
6        "content": "Sono 3 euro e 50"
7      }
8    },
9    "sessionAttributes": {
10     "4": "K",
11     "3": "G",
12     "2": "D",
13     "1": "B",
14     "0": "A"
15   }
16 }
```

Listing 6.11: Risposta della funzione AWS Lambda alla terza e ultima interazione

Alla ricezione di questo comando, Lex restituirà all'utente il messaggio "Sono 3 euro e 50" chiudendo la conversazione, ritenendo quindi l'interazione con l'utente completata, senza aspettarsi ulteriori risposte.

Capitolo 7

Implementazione della soluzione

Lo scopo di questo capitolo è descrivere l'implementazione delle funzionalità più significative dell'applicativo sviluppato, con un focus particolare sulle scelte che si sono dovute compiere.

7.1 Backend

7.1.1 Struttura del database

Nella sezione 6.2.3 si è descritta la scelta di utilizzare *MongoDB* come DBMS. *NestJS* fornisce degli strumenti per interagire con il driver *mongoose* in maniera più integrata. Quest'ultimo è una libreria scritta in *JavaScript* che fornisce una soluzione di modellazione dei dati basata su *schema*.

Uno *schema* è un particolare tipo di classe che determina la struttura di un documento, compresi eventuali vincoli sui valori e funzioni di validazione da utilizzare. Gli strumenti offerti da *NestJS* permettono di realizzare gli *schema* tramite semplici POJO. Verranno ora discusse le implementazioni degli *schema* relativi ai moduli `Chatbots` e `Answers`.

7.1.1.1 Memorizzazione delle informazioni di un *chatbot*

Per salvare le informazioni relative ad un *chatbot* si è optato per la realizzazione di un database con struttura annidata, in stile "matrioska", che è illustrata nella figura 7.1. Ogni suo elemento rappresenta uno *schema*, tra i quali si distinguono:

- `Chatbot`, che contiene le informazioni generali del *chatbot*, quali il nome, l'immagine ed il riferimento all'utente proprietario;
- `Knowledge`, ovvero la base di conoscenza del bot: indica semplicemente la lista dei temi trattati dal *chatbot*. Sebbene si sarebbe potuto evitare questo incapsulamento, aggiungendo la lista dei temi direttamente nella *collection* `Chatbot`, l'astrazione della base di conoscenza in una struttura separata permette l'eventuale implementazione futura di nuove funzionalità, come la condivisione della base di conoscenza tra più *chatbot*, senza dover riprogettare tutta la base di dati;
- `Theme` include tutte le informazioni in merito ad un tema, come il nome e la lista dei bisogni ad esso collegati;
- `Need` rappresenta un bisogno, descritto tramite il nome, la lista delle domande che fungono da *sample utterance* e la risposta (`answer`) da ritornare all'utente;

- `Question`, che contiene il testo di una singola *sample utterance*;
- `Answer`, ovvero la risposta collegata al bisogno, la cui implementazione verrà discussa nel dettaglio in 7.1.1.2.

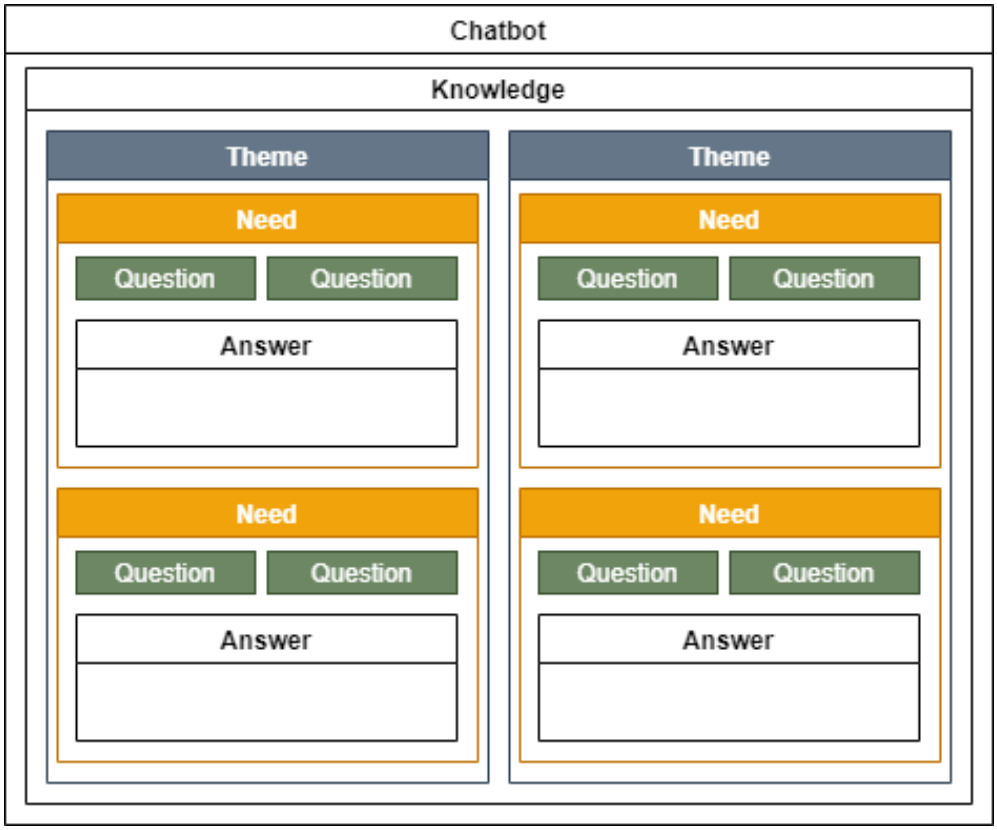


Figura 7.1: Rappresentazione a "matrioska" della struttura del database

7.1.1.2 Risposte al bisogno

Il salvataggio delle risposte ad un bisogno sul database ha richiesto l'utilizzo di particolari accortezze in fase di implementazione. Il problema principale è che esistono più tipologie di risposta, le cui funzionalità sono state incapsulate nei moduli `SimpleTextAnswer` (risposta testuale semplice), `ImageAnswer` e `VideoAnswer` (risposte di tipo multimediale) e `OptionAnswer` (che rappresenta una possibile scelta nel "Dipende Da"). Implementando uno *schema* differente per ogni tipologia di risposta sarebbe stato impossibile avere un unico attributo `answer` nello *schema* `Need` che potesse fare riferimento ad ognuna di esse. Infatti, quando si definisce un attributo di tipo *riferimento* come lo è `answer`, è necessario impostare la *collection* in cui sono presenti gli oggetti riferiti. Tuttavia in questo caso, avendo quattro *schema* differenti, essi verranno salvati in quattro *collection* differenti.

Per questo motivo si è dovuto ricorrere all'implementazione di una gerarchia di classi, sfruttando il meccanismo dell'ereditarietà messo a disposizione da *mongoose*, chiamato *model discriminator*. Questa funzionalità permette di salvare in un'unica *collection* degli oggetti

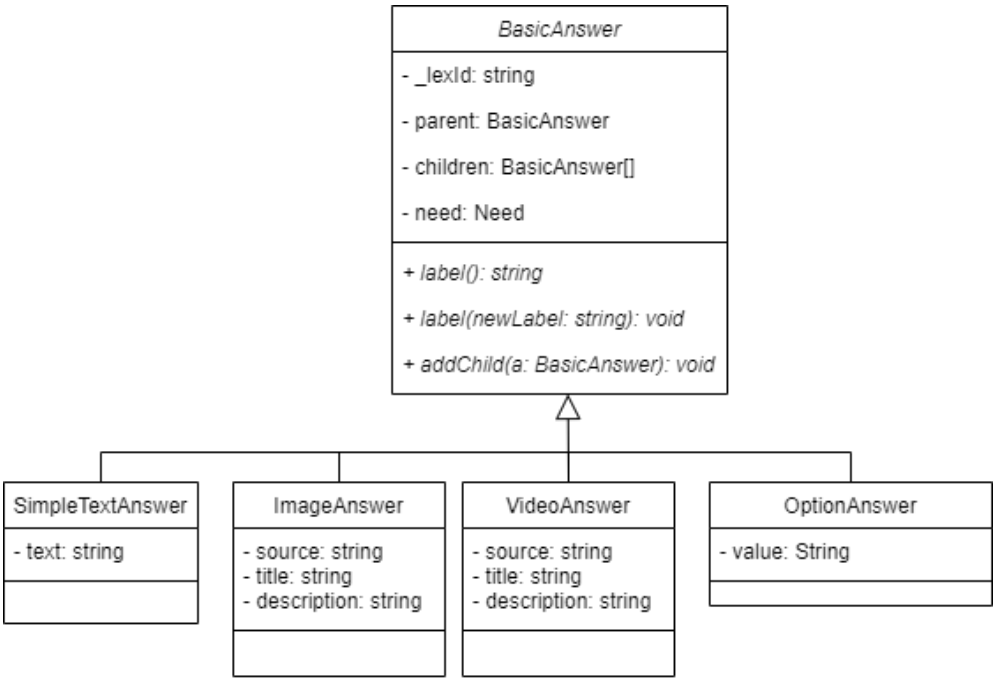


Figura 7.2: Illustrazione della gerarchia utilizzata per implementare il salvataggio delle risposte ai bisogni.

aventi solo un sottoinsieme dei loro attributi in comune, grazie all'utilizzo di un particolare campo nel modello che permette di associare ogni elemento allo *schema* corretto.

In figura 7.2 è illustrato lo schema della gerarchia utilizzata per l'implementazione delle risposte ai bisogni. Si distinguono le seguenti classi:

- `BasicAnswer`, classe base astratta, che contiene gli attributi comuni a tutte le risposte, come il riferimento al bisogno di appartenenza, quello alla risposta "padre" e alle risposte "figlie". Questi ultimi due attributi si sono resi necessari poiché un elemento di questa *collection* può far parte dell'albero di decisione di una risposta di tipo "Dipende Da", quindi è necessario tener traccia dei nodi padre e figli;
- `SimpleTextAnswer`, una risposta di testo semplice;
- `ImageAnswer` e `VideoAnswer` sono le risposte di tipo multimediale, che contengono il riferimento all'elemento multimediale, il suo titolo e una sua breve descrizione;
- `OptionAnswer` indica una delle possibili scelte di un nodo decisionale. In pratica, nell'esempio dell'intento *NuovoOrdine* riportato in figura 6.3, gli elementi di tipo `OptionAnswer` presenti sono "Pizza", "Pastasciutta", "Margherita", "Marinara", "Ragù" e "Pesto".

7.1.2 Implementazione della funzione Lambda

Nell'implementazione della funzione AWS Lambda , il cui scopo è stato descritto nella sezione 6.3.3.2 sono stati riscontrati i seguenti problemi:

- La funzione avrebbe dovuto avere accesso al database. Infatti, il calcolo della risposta da restituire a Lex avviene in base ad un ragionamento (già ampiamente spiegato in

sezione 6.3.3.2) sul flusso di conversazione, che risiede fisicamente nella *collection* `basicanswers` del database. Pertanto sarebbe necessario interfacciare, oltre che al backend dell'applicazione, anche la funzione AWS Lambda con *MongoDB*. Ciò comporterebbe la duplicazione di tutto il codice relativo alla connessione e alla gestione dei record nel database, rendendo più complicato effettuare eventuali modifiche (poiché ci sarebbero due *codebase* da aggiornare, quella del backend e quella della funzione Lambda);

- A causa di come vengono gestiti i permessi in AWS, essere creata un'istanza della funzione lambda per ogni account AWS in cui si intende importare un *chatbot* creato dall'applicazione oggetto di questa tesi. Questo comporta che se si volesse aggiornare la logica di calcolo di una risposta, si dovrebbero modificare tutte le istanza della funzione AWS Lambda installate, ovvero una per ogni utente dell'applicazione.

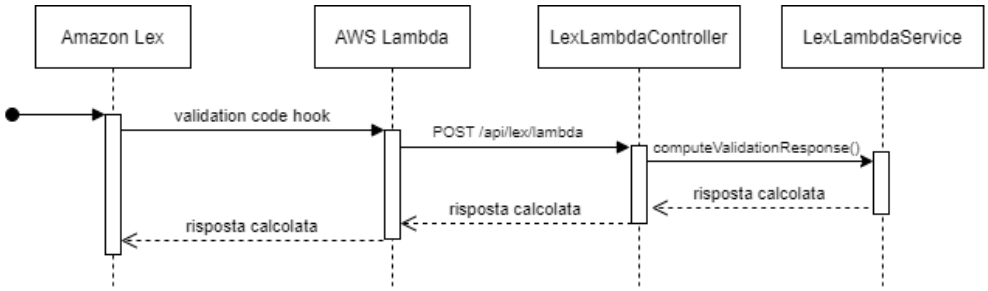


Figura 7.3: Diagramma di sequenza che illustra l'interazione tra Lex, funzione Lambda e backend dell'applicazione

Queste motivazioni hanno portato alla scelta di creare una funzione AWS Lambda che si limiti a fare da proxy tra Amazon Lex ed il backend dell'applicazione. Uno schema della sequenza delle interazioni è riportato in figura 7.3. Tutto ciò si è tradotto in una funzione AWS Lambda dal codice molto semplice e breve, letteralmente dieci righe.

Tutta la logica viene quindi gestita dalla classe `LexLambdaService`, la cui struttura è mostrata nella figura 7.4. Il suo scopo è prevalentemente quello di implementare l'algoritmo di decisione descritto in sezione 6.3.3.2 attraverso i seguenti metodi:

- `computeFulfillmentResponse`, il cui compito è calcolare la risposta proveniente da una chiamata alla funzione AWS Lambda in fase di *fulfillment*. Questa sarà la risposta finale, ovvero quella che chiuderà il dialogo e verrà calcolata percorrendo l'albero del flusso di conversazione fino ad arrivare ad uno slot;

- `computeValidationResponse`, che viene chiamata per calcolare l'output della funzione AWS Lambda quando quest'ultima viene invocata tramite il *dialog codehook*, ovvero dopo ogni interazione dell'utente con il *chatbot*.

- `followAnswerPath` viene utilizzato dalla funzione `completeFulfillmentResponse` per percorrere l'albero del flusso di conversazione coerentemente con le interazioni avute con l'utente;

- `getChoosenOption` serve a capire, dato uno slot valorizzato, quale risposta si è scelta. In pratica ritorna l'istanza di `OptionAnswer` corrispondente alla scelta svolta dall'utente. Viene utilizzata all'interno della funzione `followAnswerPath` nel caso in cui il nodo da analizzare sia di tipo *decisionale*;

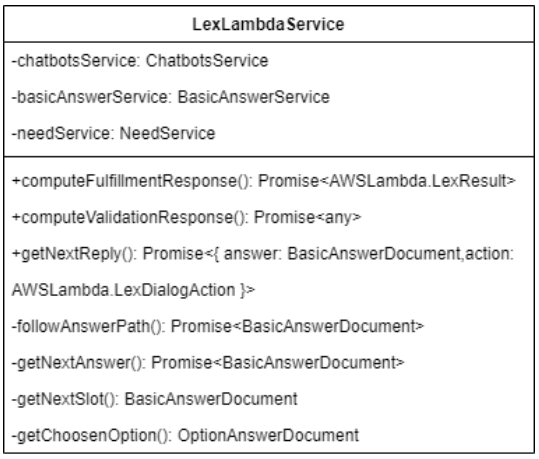


Figura 7.4: Struttura della classe `LexLambdaService`, responsabile della gestione della logica di calcolo della risposta.

- `getNextReply` si occupa di calcolare l'azione successiva che il *chatbot* dovrà svolgere in base ai valori di `sessionAttributes` e `slots` passati in input. É quindi la funzione che incapsula la logica di computazione vera e propria: determina la `dialogAction` da restituire a Lex;

- `getNextAnswer` è una funzione ausiliaria, utilizzata in `getNextReply`, che ritorna, se esiste, il nodo successivo all'ultimo visitato nel flusso conversazionale. In pratica serve a calcolare la risposta prevista in relazione al contesto conversazionale corrente;

- `getNextSlot`, anch'essa funzione ausiliaria di `getNextReply`, ritorna l'istanza di una risposta (sottoclasse di `BasicAnswer`) che corrisponde al primo slot non ancora valorizzato. In pratica ritorna la domanda successiva da porre all'utente, se esiste: in tal caso `getNextReply` restituirà una `dialogAction` di tipo `EliticSlot`.

7.2 Frontend

Lo scopo di questa sezione è mostrare come è stato effettivamente implementato il frontend, evidenziando e motivando le differenze presenti rispetto al *mockup* fornito dall'azienda.

7.2.1 Login e registrazione

La prima schermata che si è dovuto implementare, completamente assente nel *mockup* , è stata quella di login e registrazione: la piattaforma è intrinsecamente multiutente. In ogni caso, visto che il contributo innovativo da essa fornita riguarda la procedura di addestramento di un *chatbot* e non sicuramente la modalità di autenticazione, la schermata di login, visibile in figura 7.5 è molto semplice. Presenta due campi di testo, uno per la mail e uno per la password, unitamente a due bottoni, *Login* e *Registrati*, che sono collegati all'omonima funzionalità. Nella realizzazione della schermata comunque si è cercato di mantenere una certa coerenza stilistica con la UI della piattaforma. Infatti:

- Sfondo e colori sono stati ripresi dalle schermate del tutorial;

Figura 7.5: Implementazione del login e registrazione

- L'aspetto della *card* presenta bordi arrotondati e uno sfondo differenziato per l'intestazione;

- I campi di testo ed i bottoni utilizzano bordi ben evidenziati ed arrotondati.

7.2.2 Homepage

L'aspetto della schermata di homepage realizzata è riportato in figura 7.6. La prima differenza sostanziale riguarda il menu di navigazione. Il design originale (figura 5.1) prevedeva una navigazione tramite sidebar, mentre alla fine si è optato per un menu a popup. Questo ha permesso un notevole risparmio di spazio, a beneficio della visualizzazione del contenuto presente nella parte centrale della schermata. Sebbene l'homepage non sia molto densa di contenuto, questa tipologia di navigazione ha permesso di ottimizzare lo spazio soprattutto nelle schermate di addestramento del *chatbot* (ovvero nel "tutorial"), ed è stata adottata anche nella homepage per mantenere una coerenza nell'interfaccia.

Un altro elemento completamente assente nel *mockup* che però è stato necessario realizzare è la lista di tutti i *chatbot* presenti. Ogni elemento della lista corrisponde ad un *chatbot* e ne viene visualizzato il nome, l'elenco dei temi trattati e il numero di bisogni associati a ciascuno di essi. Sono inoltre presenti tre pulsanti che consentono rispettivamente la modifica, l'eliminazione ed il download del *chatbot*.

Infine l'ultimo cambiamento significativo è dato dalle opzioni fornite alla pressione del tasto "+" presente in basso a destra. Nel *mockup* , premendo quel tasto erano presenti due azioni: effettuare la procedura guidata o procedere all'addestramento in modo più "libero". In fase implementativa però si è valutato di eliminare la duplice scelta in favore dell'avvio di una procedura semi-guidata: gli step sono sempre presenti ma la guida, ovvero il tutorial vero e proprio, appare solo la prima volta che si utilizza l'applicazione per rimanere poi chiusa (ma richiamabile) durante le interazioni successive. Al posto dell'opzione rimossa è stata inserita la funzionalità di upload del file JSON esportato da Lex, in modo da poter importare dei *chatbot* già pre-esistenti. L'opzione di upload di una base di conoscenza esistente in formato `xlsx` è stata spostata nella pagina di inserimento della prima domanda, descritta in sezione 7.2.3.2.

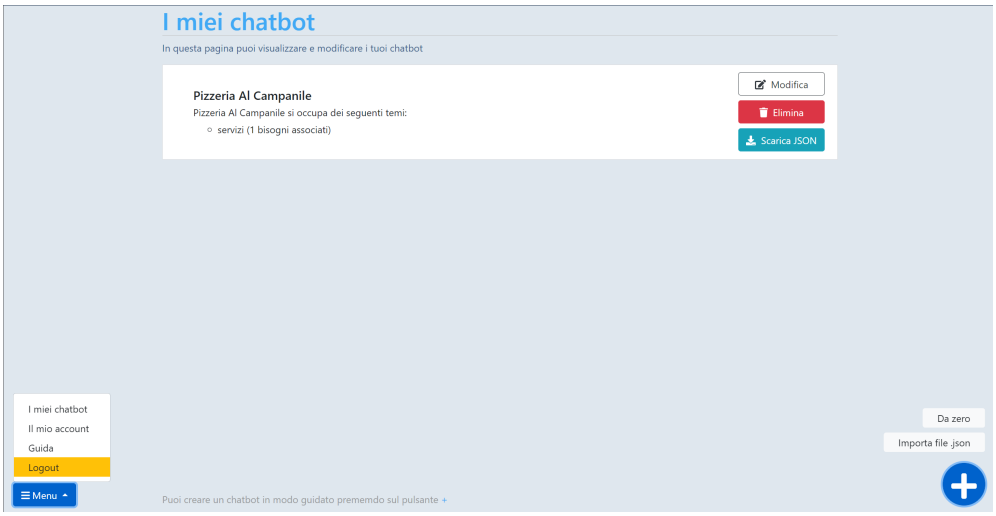


Figura 7.6: Implementazione dell'homepage

7.2.3 Wizard di addestramento

Il layout della pagina del wizard di addestramento è quello che ha subito maggiori modifiche. Implementando il layout originale, ovvero quello presente nel prototipo, ci si è accorti di quanto poco spazio sarebbe rimasto per la rappresentazione "insiemistica" della base di conoscenza del *chatbot*, problema ancora più evidente nel caso di bot complessi, ovvero con tanti temi e bisogni. Per questo motivo:

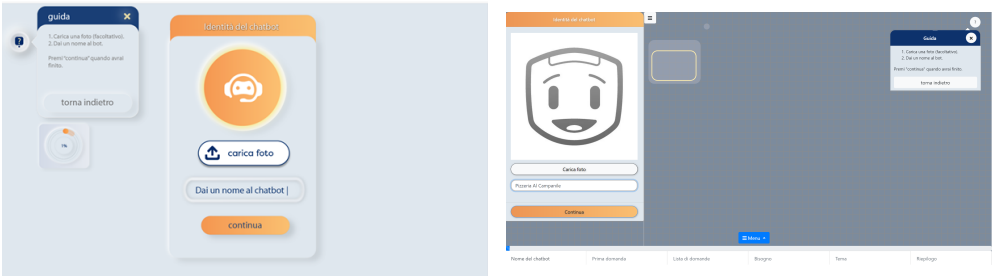
- Il tasto "?" che permette l'apertura della guida contestuale è stato reso flottante, quindi non più posizionato in un'area dello schermo riservata. Prendendo come esempio la figura 5.2 infatti si può notare come la guida e la barra di avanzamento occupino praticamente inutilmente una considerevole sezione orizzontale dello schermo, rubando spazio alla rappresentazione della base di conoscenza;
- La barra del progresso è stata spostata in basso, realizzando anche dei "segnaposto" per ogni step della procedura guidata, consentendo una navigazione più rapida all'interno di esso;
- La *card* in cui sono contenuti gli elementi che contraddistinguono i vari step è stata spostata all'interno di un pannello collassabile, in modo da poter essere all'occorrenza nascosta e permettere una più ampia visione della rappresentazione della base di conoscenza. Inoltre proprio a quest'ultima sono state fatte ulteriori modifiche, come l'inserimento dei pulsanti necessari a creare, modificare ed eliminare bisogni, temi e domande, oltre alla possibilità di spostare le domande da un bisogno ad un altro e i bisogni da un tema ad un altro semplicemente con un *drag and drop*.

Tutte queste modifiche verranno ora discusse più nel dettaglio via via che verranno illustrate le varie schermate coinvolte.

7.2.3.1 Identità del *chatbot*

La schermata di inserimento dell'identità del *chatbot* è rappresentata in figura 7.7b. Qui si possono notare tutte le modifiche esposte precedentemente:

- La guida risulta in una posizione diversa rispetto alla figura 7.7a;
- La barra del progresso si trova in basso ed assume l'aspetto di una barra di navigazione: cliccando sul nome di uno step si verrà condotti ad esso. Questa opzione è disponibile però solo in modalità di modifica del *chatbot*, non durante il tutorial iniziale, poiché le precondizioni per accedere ad ogni step del wizard potrebbero non essere soddisfatte. Ad esempio, non sarebbe possibile accedere alla schermata di modifica del tema se prima non si è definito almeno un suo bisogno;
- La *card* "Identità del *chatbot*" è stata inserita all'interno di un pannello collassabile tramite il pulsante con l'icona del menù ad hamburger posta in alto;
- Il pulsante del menù è stato spostato nella parte centrale della schermata in modo che non si sovrapponga al pannello laterale contenente la card.



(a) Design originale

(b) Schermata implementata

Figura 7.7: Confronto tra mockup ed implementazione della schermata di identità del *chatbot*

7.2.3.2 Prima domanda

La schermata di inserimento della prima domanda del *chatbot* è riportata in figura 7.8. Le modifiche rispetto al prototipo sono le stesse già discusse in 7.2.3.1.

Tuttavia è necessario fare una precisazione in merito al caricamento del file. L'azienda Primo Round ha fornito un esempio di file `xlsx` contenente una base di conoscenza (il cui estratto viene riportato in figura 7.9, con le seguenti colonne:

1. Numero della domanda;
2. Nome del tema;
3. Testo della domanda;
4. Testo della risposta.

Il caricamento di tale file in questa schermata provocherà l'aggiornamento della rappresentazione della base di conoscenza presente al centro, nonché l'uscita dalla modalità di tutorial in modo da permettere una modifica più libera del *chatbot*. L'uscita dalla modalità tutorial consiste nello sblocco della navigazione tramite i pulsanti posti sotto la barra del progresso, oltre che alla comparsa dei bottoni per aggiungere, modificare ed eliminare temi e bisogni.

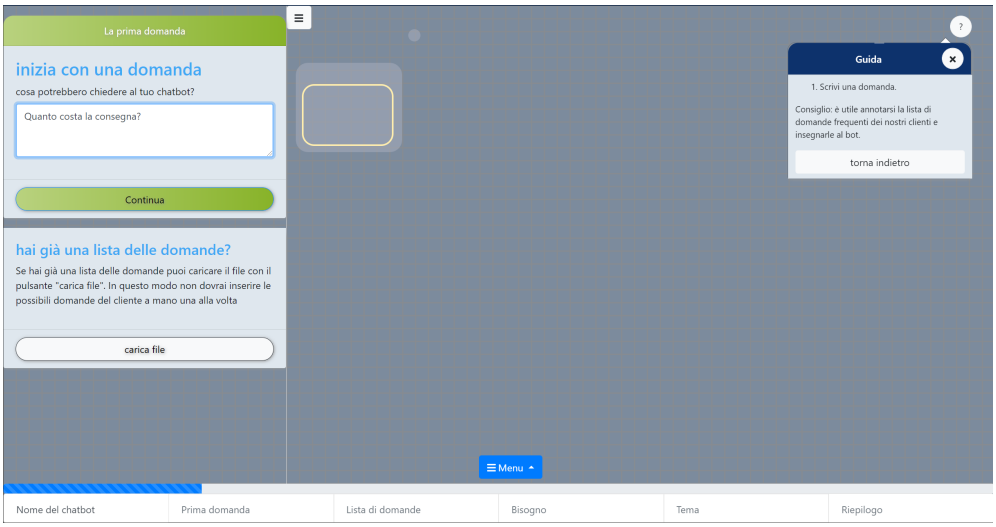
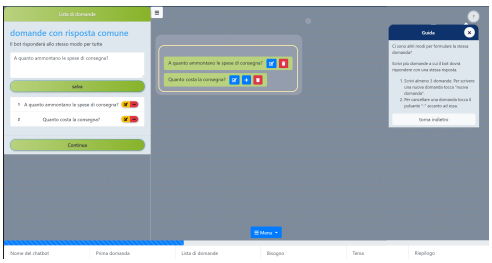


Figura 7.8: Implementazione della schermata di inserimento della prima domanda

		C	D
		Domande	**Risposte**
1	P	**Cosa è PIPPO?**	PIPPO è un dispositivo medico di classe II b utile per il trattamento della sindrome metabolica e il riequilibrio di parametri metabolici alterati quali livelli di colesterolo, trigliceridi, glucosio e valori di circonferenza addominale.
2	P	**Esiste un prodotto per la sindrome metabolica?**	Si, esiste PIPPO, un dispositivo medico di classe II b indicato per il trattamento della sindrome metabolica e il riequilibrio di parametri metabolici alterati quali livelli di colesterolo, trigliceridi, glucosio e valori di circonferenza addominale.

Figura 7.9: Struttura del file contenente una base di conoscenza esistente

(a) Design originale



(b) Schermata implementata

Figura 7.10: Confronto tra mockup ed implementazione della schermata di inserimento della lista delle domande

7.2.3.3 Lista delle domande

Lo step del tutorial che permette all'utente di inserire la lista delle domande collegate al bisogno è mostrato in figura 7.10b. Rispetto al *mockup* oltre alle modifiche di posizionamento degli elementi già ampiamente discusse, sono stati inseriti i pulsanti che permettono di modificare o eliminare ogni domanda, sia nella visualizzazione insiemistica che nell'elenco delle domande sul lato sinistro dello schermo. Infatti testando la versione concepita dal *mockup* ci si è accorti come risultasse poco chiara la funzionalità di modifica delle domande, inizialmente concepita tramite un click sopra il testo della domanda stessa (si veda la figura 7.10a). Inoltre l'aspetto grafico degli elementi della lista, ovvero rettangoli dal bordo arrotondato distanziati tra loro, rendeva troppo dispersiva la rappresentazione, soprattutto in caso di schermo piccolo o di elenco lungo.

7.2.3.4 Nome del bisogno

Figura 7.11: Implementazione della schermata di inserimento del nome del bisogno, istanziata all'esempio della "Pizzeria Al Campanile"

La schermata di inserimento del nome del bisogno, riportata in figura 7.11, non ha subito modifiche particolari Rispetto al *mockup* ad eccezione di quelle in merito al layout globale già discusse in precedenza. L'unico cambiamento è stato rendere la *card* più corta, ovvero evitare che si estendesse inutilmente per tutta l'altezza dello schermo.

7.2.3.5 Selezione della risposta del bisogno

La pagina di selezione delle risposte di un bisogno, riportata in figura 7.12, ha subito una serie di modifiche. Nel *mockup* erano presenti tre tipi di risposta: testuale, lineare e "dipende da". Il tipo di risposta lineare, all'inizio concepito come un lungo testo formattato, è stato sostituito con la tipologia di risposta "elemento multimediale". Gli altri cambiamenti riguardano l'implementazione delle singole opzioni di risposta, che vengono ora descritte nel dettaglio

7.2.3.5.1 Dipende da Riguardo al "Dipende Da", è stata inserita un'anteprima dell'albero del flusso conversazionale al posto dello spazio vuoto con sfondo quadrettato presente nella figura 5.7. Ciò è stato fatto sempre in ottica di ottimizzazione dello spazio disponibile, cercando

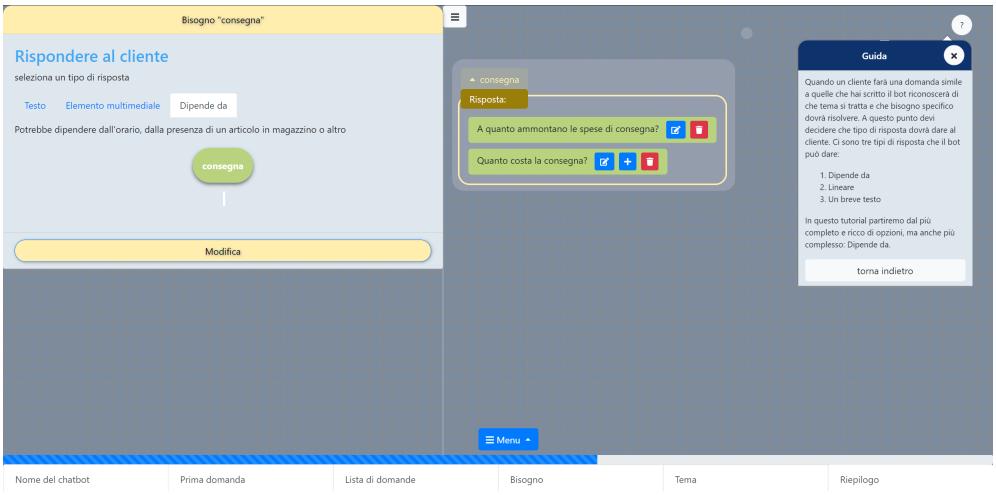


Figura 7.12: Implementazione della schermata di selezione del tipo di risposta ad un bisogno

di fornire all'utente un'informazione quanto più significativa possibile. Nella figura 7.12 l'albero appare formato da un nodo solo (che è il bisogno, e rappresenta il punto di ingresso del ragionamento), in quanto la risposta non è ancora stata definita. Al termine dell'inserimento tuttavia il contenuto della *card* sarà quello riportato in figura 7.13.

Per quanto riguarda l'inserimento e la modifica di questa tipologia di risposta, l'interfaccia è riportata in figura 7.14b. Si possono notare delle differenze rispetto al *mockup* (figura 7.14a), come:

- Il nodo radice ora contiene il nome del bisogno,;
- I nodi "opzione", cioè quelli con sfondo verde che contengono le scelte "Si" e "No", hanno il testo modificabile, in modo da poter implementare facilmente altre tipologie di scelte binarie, come ad esempio "Ok" e "Annulla", semplicemente editando il testo delle opzioni;
- Nei nodi di scelta, ovvero quelli con sfondo arancione (e nell'immagine 7.14b) testo "Sei di Padova") è stato inserito un tasto che permette di aggiungere una nuova opzione di risposta in maniera rapida. Ciò permette in modo rapido di aggiungere nuovi rami al flusso conversazionale senza dover rifare da capo interi sotto-alberi.

7.2.3.5.2 Risposta di testo semplice

La schermata di risposta testuale semplice, riportata in figura 7.15, non ha subito modifiche particolari.

7.2.3.5.3 Elemento multimediale

L'inserimento della risposta di tipo "lineare" è stato modificato in "elemento multimediale" per una mancanza di supporto da parte di Lex. Un elemento multimediale può essere di due tipologie: immagine e video. Una panoramica della modalità di inserimento è riportata in figura 7.16.

7.2.3.6 Nome del tema

Anche la schermata di inserimento del nome del tema ha subito le stesse modifiche di quella di inserimento del nome del bisogno. In ogni caso, la versione finale è riportata in figura 7.17.

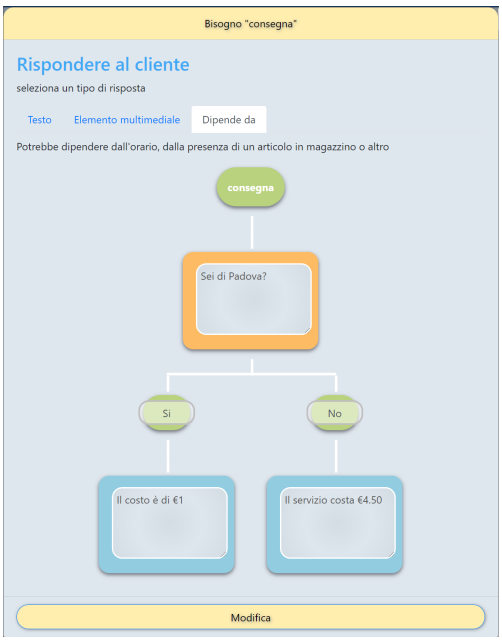
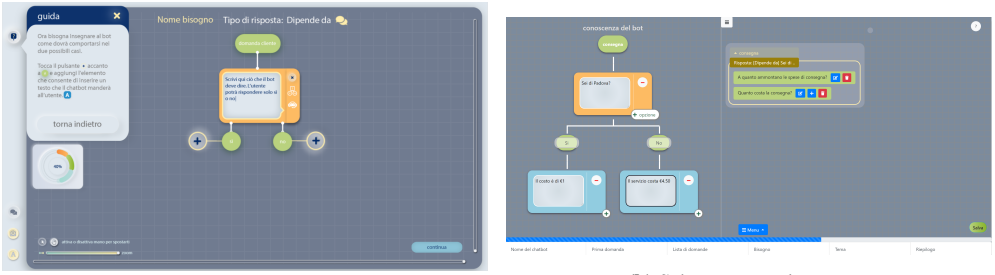


Figura 7.13: Dettaglio della schermata del tipo di risposta "dipende da" caso in cui sia già stata inserita.



(a) Design originale

(b) Schermata implementata

Figura 7.14: Confronto tra una schermata di modellazione di una risposta di tipo "dipende da" presente nel *mockup* e la versione implementata

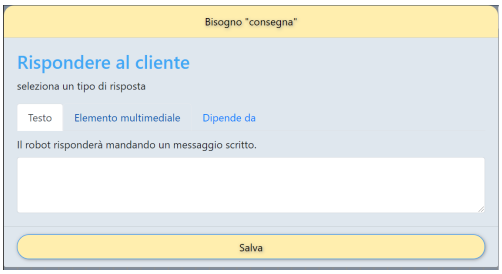


Figura 7.15: Implementazione della schermata di inserimento di un tipo di risposta testuale

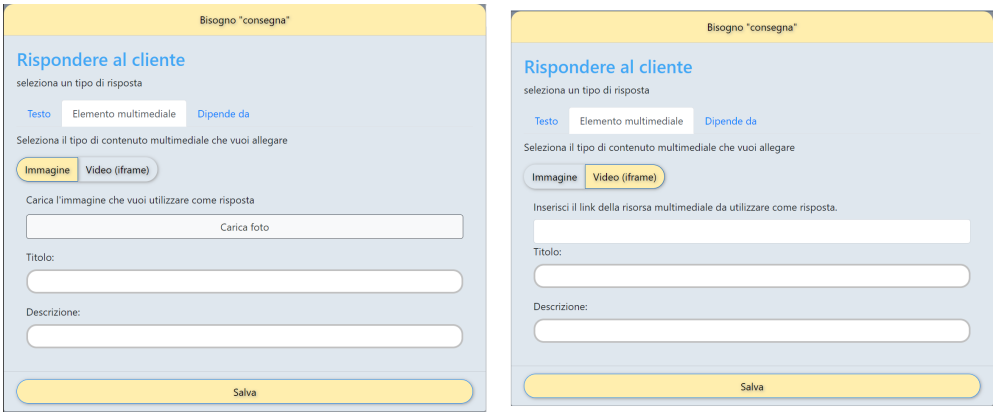


Figura 7.16: Implementazione della schermata di inserimento di un tipo di risposta multimediale (immagine o video)

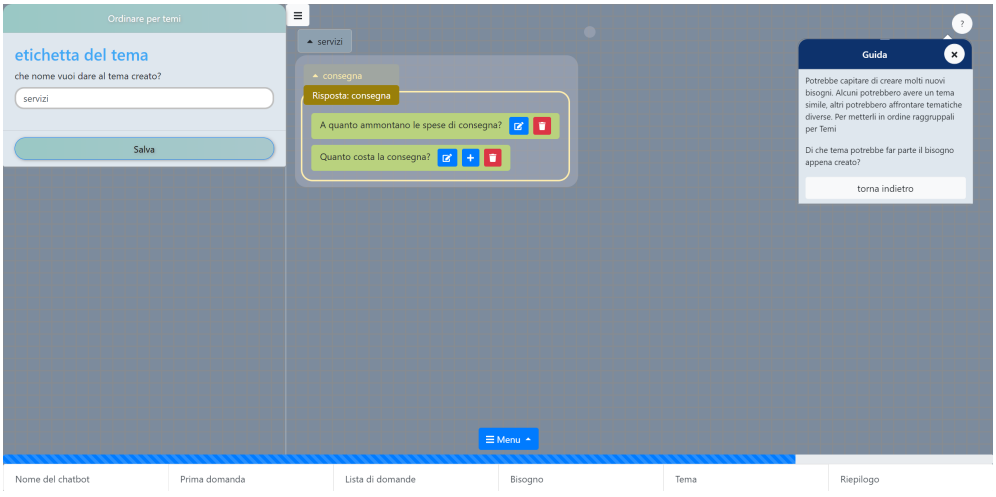


Figura 7.17: Implementazione della schermata di inserimento del nome del tema, istanziata all'esempio della "Pizzeria Al Campanile"

7.2.3.7 Riepilogo finale

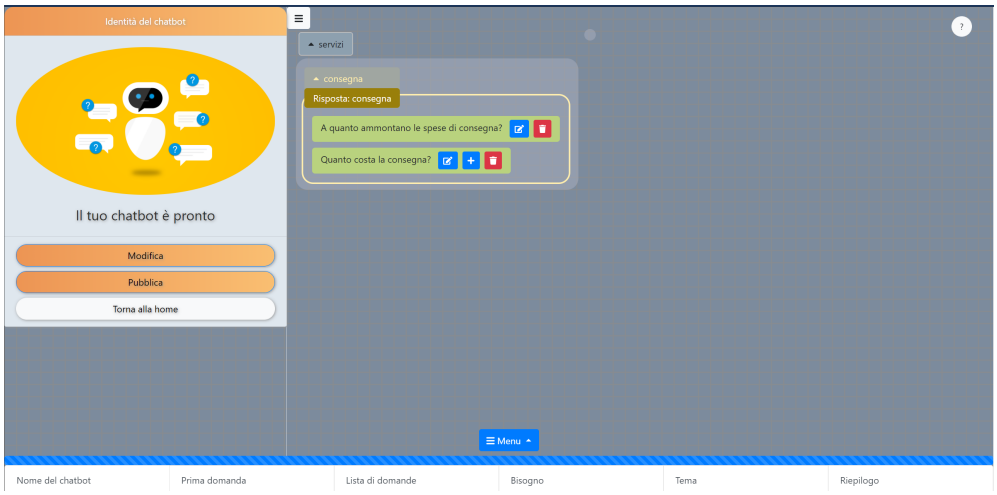


Figura 7.18: Implementazione della schermata riepilogo mostrata al termine dell'addestramento, istanziata all'esempio della "Pizzeria Al Campanile"

Rispetto al *mockup* lo step finale del tutorial ha subito due modifiche sostanziali:

- La rimozione delle "stelline" come punteggio dell'addestramento, dovuto all'impossibilità di trovare una metrica semplice ed oggettiva sulla base di cui calcolarle;
- Lo spostamento del pulsante "Torna alla home" che alla pressione conduce l'utente alla homepage (quella della figura 7.6) fuori dal popup della guida per renderlo sempre visibile.

Infine, cosa non specificata nel *mockup* , la pressione del tasto "pubblica" provoca il download del file JSON con la definizione del *chatbot*, che andrà poi importato in Lex.

7.2.4 Dettaglio della rappresentazione insiemistica della base di conoscenza

La rappresentazione insiemistica della base di conoscenza ha subito delle modifiche abbastanza importanti:

- É stata data una forma rettangolare all'insieme che contiene temi e bisogni, sia per una questione di ottimizzazione dello spazio che per una maggior semplicità implementativa;
- Sono state inseriti, a fianco del nome dei temi e dei bisogni, dei pulsanti a forma di triangolo che permettono di espandere o collassare l'insieme, dando la possibilità di avere una rappresentazione all'occorrenza più compatta soprattutto nel caso di una base di conoscenza di dimensione significativa, come quella della figura 7.19;
- Sotto il nome di ogni bisogno è stato inserito il riepilogo della risposta ad esso associata all'interno di un rettangolino marrone;
- Sono stati inseriti pulsanti che permettono di creare, modificare o eliminare ogni elemento, quali domande, bisogni, risposte e temi. Questi saranno disponibili solo al termine della procedura guidata, ovvero in fase di modifica di un *chatbot* esistente.

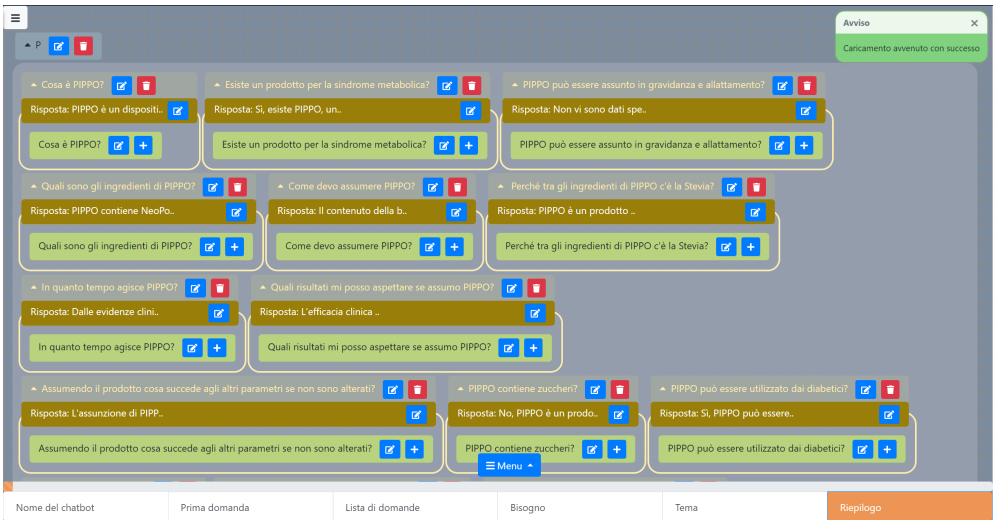


Figura 7.19: Esempio di rappresentazione di una base di conoscenza complessa

Capitolo 8

Conclusioni

L'applicativo realizzato rappresenta senz'altro un *proof-of-concept* che dimostra come l'addestramento di un *chatbot* in maniera *code-free* sia possibile. Delle soluzioni analoghe sono già presenti sul mercato, ed alcune di esse sono state anche analizzate all'inizio di questo elaborato. Tuttavia, l'approccio proposto si differenzia per **come** il *chatbot* viene creato e per **cosa** viene effettivamente creato.

Come viene creato Per il design e la realizzazione del *chatbot*, la piattaforma usa un approccio interamente guidato e visuale, senza mai richiedere l'inserimento di codice, pseudocodice o, ancora più in generale, l'utilizzo di conoscenze informatiche specifiche. I termini tecnici chiave, come il concetto di *intento* o di *istruzione condizionale*, vengono spiegati all'utente tramite delle metafore (rispettivamente il *bisogno* e il *dipende da*). Inoltre l'intero procedimento è realizzato sotto forma di wizard, ovvero una procedura che guida l'utente lungo certi passaggi. Ciò, se da un lato può penalizzare un utente avanzato, in quanto non sono presenti "scorciatoie", sicuramente risulta molto utile ad un utente alle prime armi, poiché dice esattamente ogni volta cosa bisogna fare e quando.

Che cosa viene creato L'applicazione non integra nessun motore di NLP o NLU quindi nessuna intelligenza artificiale. Al contrario, l'approccio usato consiste nell'affidarsi interamente a piattaforme esterne leader nel settore per eseguire il *chatbot*. Ciò ha permesso di eliminare il problema di creare e mantenere un'intelligenza artificiale "proprietaria", processo molto costoso soprattutto in termini di dati necessari per l'addestramento. Affidandosi ad una piattaforma esterna come *Lex*, questo compito viene delegato ad *Amazon*, che possiede sicuramente molte più risorse, sia in termini di dati necessari che di potenza computazionale richiesta, rispetto quelle a nostra disposizione. L'intelligenza artificiale di *Lex* è anche alla base dell'assistente di *Amazon*, *Alexa*: questo è causa e conseguenza di un miglioramento continuo a cui viene sottoposta, aumentando la precisione del riconoscimento praticamente di giorno in giorno.

8.1 Sviluppi futuri

L'applicazione realizzata presenta senz'altro dei margini di miglioramento in termini di funzionalità offerte. Innanzitutto potrebbero essere integrate altre piattaforme sulle quali esportare il *chatbot*, come *IBM Watson* o *Google Dialogflow*. Per come è strutturata l'applicazione, sarebbe sufficiente semplicemente aggiungere un nuovo modulo per ogni piattaforma, incaricato della

traduzione del *chatbot* nel formato di *Watson* o *Dialogflow*.

Un altro possibile sviluppo potrebbe essere introdurre delle tecniche di *Named Entity Recognition* per riconoscere automaticamente eventuali entità presenti nelle *utterance* dell'utente. Queste consentirebbero di creare un'interazione più naturale con il bot, in quanto permetterebbero di estrarre più informazioni da una singola interazione con l'utente. Nell'esempio della pizza presentato in sezione 6.3.3.2, l'*utterance* utilizzata dall'utente potrebbe includere anche il tipo di pizza da ordinare, del tipo: "Vorrei ordinare una pizza margherita". In questo caso il *chatbot* dovrebbe essere in grado di estrarre i valori delle entità `PietanzaEntity` e `TipoDiPizzaEntity` in maniera automatica, senza doverli chiedere all'utente.

Infine si potrebbero implementare delle tecniche di miglioramento continuo delle capacità del *chatbot*, chiamate "gestione della retroazione". Queste consistono nell'estrapolare dalle conversazioni che il *chatbot* ha con gli utenti le espressioni da esso non riconosciute, consentendo così ad un operatore umano di assegnarle agli intenti corretti, addestrando di fatto il *chatbot*. In altre parole, le espressioni non riconosciute vengono etichettate manualmente e diventano dati di training per l'intelligenza artificiale che esegue il *chatbot*.

Glossario

Adobe XD Software di progettazione di UI e UX per dispositivi mobili, siti web e applicazioni desktop.

AJAX Asynchronous JavaScript and XML, tecnica di realizzazione di applicazioni web che consente di richiedere informazioni ad un server (backend) senza dover ricaricare l'intera pagina..

API Application Programming Interface.

AWS Amazon Web Services è un ramo del gruppo Amazon che offre servizi cloud, che vanno dall'hosting di siti web allo storage online, fino all'affitto di potenza computazionale.

AWS Lambda Piattaforma di calcolo serverless. Permette di eseguire del codice su cloud pensando automaticamente all'allocazione delle risorse richieste. Attualmente i linguaggi di programmazione supportati sono C#,Go, Java, Node.js, PowerShell, Python e Ruby..

CSS Cascading Style Sheet, linguaggio per la definizione dello stile di una pagina web scritta in HTML..

DBMS DataBase Management System, software che consente la creazione, modifica ed interrogazione di una base di dati..

decorator Particolare tipo di dicharazione che può essere associata ad una classe, un metodo, una proprietà o un parametro. I decorator usano una notazione del tipo "@espressione", dove "espressione" è una funzione che viene invocata a runtime..

Dependency Injection Design pattern della Programmazione orientata agli oggetti, ideato per diminuire le dipendenze fra componenti in un'architettura software. L'idea alla base della Dependency Injection è quella di avere un componente esterno (assembler) che si occupi della creazione degli oggetti e delle loro relative dipendenze e di assemblarle mediante l'utilizzo dell'injection.

Express.js Express.js è un framework per Node.js che mira a semplificare la gestione delle richieste HTTP. Citando la documentazione, "Express fornisce uno strato sottile di funzionalità di base per le applicazioni web, senza nascondere le funzioni Node.js che conosci e ami." [28].

HTML Hyper Text Markup Language, linguaggio di markup per la realizzazione di pagine web..

JSON JavaScript Object Notation, formato per lo scambio di dati che utilizza una sintassi simil-javascript..

LESS Leaner Style Sheets, estensione di CSS che permette di usare variabili e mixin, oltre a consentire operazioni tra variabili (come somme/differenze o concatenazioni). Essendo un'estensione di CSS, ogni istruzione CSS è anche un'istruzione LESS (la sintassi non cambia)..

MEAN MEAN, acronimo di "MongoDB, Express.js, Angular and Node.js" è uno stack tecnolgico per lo sviluppo di Single Page Application che consiste nell'utilizzo di Node.js come backend, Express.js come libreria per la realizzazione delle API di comunicazione, Angular per la creazione del frontend e MongoDB come database NoSQL..

NLP Natural Language Processing, ramo dell'intelligenza artificiale che affronta il problema dell'elaborazione e comprensione del linguaggio naturale. .

NLU Natural Language Understanding, ramo dell'intelligenza artificiale (più specificatamente del Natural Language Processing) che affronta il problema di capire il significato di un testo scritto in linguaggio naturale.

Node.js Node.js è un runtime JavaScript progettato per creare applicazioni di rete scalabili; fa ampio utilizzo della programmazione asincrona, prestandosi così alla realizzazione di applicazioni web scalabili..

OOP Object-Oriented Programming, programmazione orientata agli oggetti.

POJO *Plain-Object JavaScript Model*, oggetto *JavaScript*che contiene solo dati. É l'equivalente di una classe che contiene solo attributi, senza specificare nessun metodo..

SPA Single Page Application, applicazione web costituita di un'unica pagina il cui contenuto cambia dinamicamente in base all'interazione con l'utente. La differenza tra una SPA e una web-application "tradizionale" è che nella SPA non c'è mai un ricaricamento completo della pagina..

SRP Single Responsability Principle, best practice della programmazione orientata ad oggetti che sancisce che ogni classe dovrebbe avere una singola responsabilità. In tal senso, se si vuole modificare una funzionalità, sarà necessario mettere mano solo alla classe/alle classi che la implementano, evitando di modificare le classi collegate..

Bibliografia

[1] K. H. Davis, R. Biddulph e S. Balashek. «Automatic Recognition of Spoken Digits». In: *The Journal of the Acoustical Society of America* 24.6 (1952), pp. 637–642. DOI: 10.1121/1.1906946. eprint: https://doi.org/10.1121/1.1906946. URL: https://doi.org/10.1121/1.1906946.

[2] Douglas C. Engelbart. «Augmenting the Human Intellect: A conceptual framework.» In: (1962). URL: https://www.dougengelbart.org/content/view/138.

[3] IBM. *IBM Shoebox*. 1962. URL: https://www.ibm.com/ibm/history/exhibits/specialprod1/specialprod1_7.html.

[4] Bruce Lowerre. «The Harpy Speech Understanding System». In: *Readings in Speech Recognition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 576–586. ISBN: 1558601244.

[5] Andries van Dam. «Post-WIMP User Interfaces». In: (1997). DOI: 10.1145/253671.253708.

[6] John Redant. *HCI Review of the Xerox Star*. 2001. URL: https://xeroxstar.tripod.com/index.html.

[7] Alan Dix et al. *Human-Computer Interaction (3rd Edition)*. USA: Prentice-Hall, Inc., 2003. ISBN: 0130461091.

[8] Steve Young Mark Gales. «Application of Hidden Markov Models in Speech Recognition». In: (2007). DOI: 10.1561/2000000004. URL: https://mi.eng.cam.ac.uk/~mjfg/mjfg_NOW.pdf.

[9] James Vincent. 2016. URL: https://www.theguardian.com/technology/2016/mar/24/tay-microsofts-ai-chatbot-gets-a-crash-course-in-racism-from-twitter?CMP=twt_a-technology_b-gdntech.

[10] Katia Moskvitch. *The Machines That Learned to Listen*. 2017. URL: https://www.bbc.com/future/article/20170214-the-machines-that-learned-to-listen.

[11] Matt Sornson. *The Modern Guide to Lead Qualification*. 2017. URL: https://clearbit.com/blog/the-modern-guide-to-lead-qualification/.

[12] Guillaume Baudart et al. «Reactive Chatbot Programming». In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 21–30. ISBN: 9781450360708. DOI: 10.1145/3281278.3281282. URL: https://doi.org/10.1145/3281278.3281282.

[13] Fran Conejos. *Conversational Interface: The Ultimate Breakdown*. 2018. URL: https://landbot.io/blog/conversational-interfaces-explained/.

[14] Patrik Jonell et al. «Fantom: A Crowdsourced Social Chatbot using an Evolving Dialog Graph». In: 2018.

[15] Venus Tamturk. *Enterprises Set Themselves Up for Success via Chatbots*. 2018. URL: https://bit.ly/3vGkvGe.

[16] Patrik Jonell et al. «Crowdsourcing a Self-Evolving Dialog Graph». In: *Proceedings of the 1st International Conference on Conversational User Interfaces*. CUI '19. Dublin, Ireland: Association for Computing Machinery, 2019. ISBN: 9781450371872. DOI: 10.1145/3342775.3342790. URL: https://doi.org/10.1145/3342775.3342790.

[17] Arthur Lacerda e Carla Rocha. «FLOSS FAQ chatbot project reuse: how to allow non-experts to develop a chatbot». In: ago. 2019, pp. 1–8. ISBN: 978-1-4503-6319-8. DOI: 10.1145/3306446.3340823.

[18] Teresa Castle-Green et al. «Decision Trees as Sociotechnical Objects in Chatbot Design». In: *Proceedings of the 2nd Conference on Conversational User Interfaces*. CUI '20. Bilbao, Spain: Association for Computing Machinery, 2020. ISBN: 9781450375443. DOI: 10.1145/3405755.3406133. URL: https://doi.org/10.1145/3405755.3406133.

[19] *ActiveChat*. URL: https://activechat.ai/.

[20] *Amazon Lex*. URL: https://aws.amazon.com/it/lex/.

[21] *Angular*. URL: https://angular.io/.

[22] *Chatfuel.com*. URL: https://chatfuel.com/.

[23] *Cleverbot*. URL: https://www.cleverbot.com/.

[24] *Documentazione di Microsoft QnA Maker*. URL: https://bit.ly/3l7wirX.

[25] *Drift.com*. URL: https://app.drift.com/.

[26] *EcmaScript 6*. URL: http://es6-features.org/.

[27] *Engati.com*. URL: https://www.engati.com/.

[28] *Express.js*. URL: https://expressjs.com/it/.

[29] *Formato di input e output della funzione Lambda per l'interazione con Lex*. URL: https://amzn.to/3907f5g.

[30] *Google Dialogflow*. URL: https://cloud.google.com/dialogflow.

[31] IBM. *Pioneering Speech Recognition*. URL: https://www.ibm.com/ibm/history/ibm100/us/en/icons/speechreco/.

[32] *Intercom.com*. URL: https://www.intercom.com/drlp/customizable-bots-biz.

[33] *Landbot.io*. URL: https://app.landbot.io/.

[34] *ManyChat*. URL: https://manychat.com/.

[35] *Microsoft QnA Maker*. URL: https://qnamaker.ai/.

[36] *MongoDB*. URL: https://www.mongodb.com/it.

[37] *Nest.JS*. URL: https://nestjs.com/.

[38] Oracle. *What is a Chatbot?* URL: https://www.oracle.com/middleeast/chatbots/what-is-a-chatbot/.

[39] *Quote di mercato degli assistenti digitali più famosi*. URL: https://www.statista.com/statistics/792604/worldwide-smart-speaker-market-share/.

[40] *Snapshot della documentazione di Dialogflow CX consultata per la fase di progettazione*. URL: https://bit.ly/3lpm86b.

[41] *Tars.com*. URL: https://hellotars.com/.

[42] *TypeScript*. URL: https://www.typescriptlang.org/.