UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze e Tecnologie

*Corso di Laurea Magistrale in Informatica*

Sistema visuale per la progettazione di flussi di dialogo per assistenti virtuali

Relatore: Prof. Stefano VALTOLINA

Controrelatore: Prof. Marco MESITI

Correlatrice: Dr.ssa Sara BONFITTO

Tesi di:

Lorenzo NERI

Matricola: 901009

Anno Accademico 2018-2019

Prefazione

Organizzazione della tesi

La tesi è organizzata come segue:

* Nel Capitolo 1, viene introdotto ed esposto l’obiettivo di questa tesi di laurea e le problematiche che propone di risolvere.
* Nel Capitolo 2, viene esposta la panoramica delle soluzioni messe a disposizione sul mercato, confrontandole tra di esse ed esponendo i punti a favore e a sfavore.
* Nel Capitolo 3, viene presentata la soluzione realizzata per raggiungere l’obiettivo su cui si basa questa tesi di laurea, includendo l’analisi effettuata a seguito del confronto con le soluzioni presenti attualmente sul mercato.
* Nel Capitolo 4, si approfondisce l’architettura del sistema implementato per la soluzione proposta.
* Nel Capitolo 5, si espone l’analisi effettuata a seguito dei test sottoposti a potenziali utilizzatori finali e i risultati ottenuti dalla soluzione proposta.

Ringraziamenti

Indice

* Capitolo 1: Pagina 5
* Capitolo 2: Pagina 7
* Capitolo 3: Pagina 8
* Capitolo 4: Pagina 16
* Capitolo 5: Pagina 38
* Bibliografia: Pagina 39

Capitolo 1

Introduzione

Un chatbot è un software che dialoga con l’utente all’interno di un contesto definito. Si potrebbe quindi definire come un assistente virtuale capace di rispondere ad una serie di richieste, ma esse devono essere ben studiate a priori in fase di progettazione del chatbot.

Per farlo in maniera ottimale è quindi necessario conoscere molto bene il contesto di utilizzo, in particolar modo il linguaggio e gli obiettivi.

Dal nome chatbot si intuisce quindi che questo assistente virtuale può essere interpellato dall’utente attraverso la chat di una piattaforma, ma sarebbe riduttivo definirlo soltanto in questo modo.

Infatti, esistono due tipologie di chatbot: la prima interagendo tramite chat e la seconda interagendo verbalmente. Esiste infatti la possibilità di interpellare il software attraverso l’uso della voce.

Da questo si capisce che con un chatbot è possibile avere una conversazione in cui il nostro interlocutore non è un essere umano, ma un robot virtuale al quale possiamo porre una serie di domande.

Esso è quindi un interlocutore sempre disponibile in qualsiasi momento della giornata, che ci può guidare e aiutare nel compiere alcune attività nella vita di tutti i giorni. Ovviamente, essendo un robot è preparato solo su alcuni argomenti sui quali può darti una serie di risposte o sui quali può dialogare.

Un chatbot ben programmato riesce comunque a condurre una conversazione attraverso un flusso di dialogo sensato e coerente, inoltre, con il machine learning riesce anche ad apprendere informazioni aggiuntive e utilizzarle in un secondo momento.

Questi robot dispongono di un’intelligenza artificiale che diventerà sempre più importante soprattutto utilizzato in certi ambiti, come ad esempio quello medico.

Ad esempio, potrebbe riconoscere in base a dei sintomi specifici che tipo di medicinale un paziente dovrebbe assumere o quali potrebbero essere le possibili cause dei sintomi.

Ovviamente per realizzare questo è sempre necessario un esperto del settore oltre che a svariati test. Potrebbe anche essere utile per velocizzare alcune attività come l’acquisto di prodotti attraverso un sito web. Parliamo quindi di robot virtuali che possono darci una mano in diversi momenti della giornata e in contesti differenti.

La realizzazione di un chatbot generalmente, impiega due figure professionali: la prima si occupa di disegnare il flusso di dialogo tra chatbot stesso e utente finale, implementando le frasi legate al contesto conversazionale, le risposte, le parole chiave di rilievo.

La seconda invece, si occupa di realizzare il software vero e proprio finalizzato all’implementazione del chatbot vera e propria.

Questo processo, richiede parecchio tempo e un continuo confronto tra le due figure prese in esame: generalmente, chi si occupa di disegnare i flussi di dialogo non ha esperienza o abilità legate alla programmazione, ha comunque conoscenze informatiche limitate che non gli permettono di realizzare software autonomamente.

Al contempo, chi ha notevoli competenze informatiche non è generalmente una figura professionale che possa realizzare flussi di dialogo in autonomia.

In diverse situazioni, siano esse legate a contesti aziendali, sanitari, di servizio pubblico o private, vi è non solo la necessità di offrire un chatbot, che permetta un’interazione nuda e cruda con un utente finale fatta di pure conversazioni testuali: è necessario permettere la creazione di promemoria, invio di messaggi, visualizzazione di contenuti multimediali e ottenimento di dati provenienti da terze parti (API, piattaforme di messaggistica, siti web, informazioni personali, codici, ecc.).

In aggiunta al contesto di dialogo, la necessità di dati e informazioni provenienti da terze parti, subentra il bisogno di offrire le utilità del proprio agente non solo in formato visivo, ma anche sonoro.

Si pensi a tutti gli utenti non vedenti o che necessitano dei servizi offerti dall’agente in situazioni in cui non possono fare un uso diretto di uno schermo: durante la guida, nel mezzo di una camminata, oppure tramite una telefonata tradizionale.

Un altro aspetto di rilievo, è legato agli errori effettuati dall’utente. In differenti situazioni, l’agente si trova impotente di fronte a frasi incomplete, con errori grammaticali, sconnesse.

Esso di fatto, per migliorare quanto più possibile l’esperienza offerta all’utente finale, deve avere a disposizione un sistema di apprendimento affinché possa imparare lui stesso dagli errori commessi dall’utente finale.

Il nostro obiettivo, è di fatto fornire un sistema di semplice utilizzo per i disegnatori di flussi di dialogo, affinché possano in totale autonomia realizzare un agente completo senza saper programmare.

Capitolo 2

Stato dell’arte

L’epoca che stiamo vivendo in questi anni, nota come “era dell’informazione”, deve il suo nome al numero sempre più crescente di utenti che desiderano svolgere le proprie attività grazie all’ausilio di un calcolatore senza vedersi costretti a diventare specialisti in materia.

Tale desiderio si sta trasformando sempre più in una domanda di mercato ad ampio spettro: c’è sempre più richiesta di ambienti di lavoro flessibili, potenti e personalizzabili in base alle esigenze, abilità e necessità di ogni singolo utente.

Non solo: c’è stata una grande evoluzione dei ruoli coinvolti nel ciclo di vita del software.

L’era dell’informazione ha portato a vedere con un punto di vista differente i computer: non più cosa possono fare i computer stessi, ma cosa possono fare le persone con essi.

Si è passati dal modello classico del produttore-consumatore a una fusione delle due figure che, in inglese, lo si identifica come “prosumer”.

Il prosumer è sì un fruitore del software, al contempo ne fa un uso ben specifico per realizzare prodotti e servizi propri.

Il mondo dell’informatica si è adattato a questo cambiamento di scenario: nel ciclo di realizzazione di un software, è diventato una necessità prestare quanta più attenzione possibile al lato umano, quindi l’esperienza dell’utente finale e le interazioni che esso ha con la soluzione offerta.

Le necessità dell’utente variano in base alla disciplina in cui esso è esperto, ma in generale si tratta di figure professionali che non hanno grandi competenze informatiche: spesso non sono nemmeno interessate ad averne.

Questa motivazione, il disinteresse degli utenti finali verso le discipline informatiche, ha creato la necessità di studiare e analizzare i metodi più opportuni per far fronte alle necessità degli utenti finali in funzione delle loro attività quotidiane.

L’utente finale nel caso di questa tesi, è l’esperto di dialoghi.

L’esperto di dialoghi, come esposto nel capitolo precedente, spesso non dispone delle competenze informatiche necessarie per realizzare il codice di un chatbot.

Allo stato attuale, prendendo come riferimenti di mercato e stato dell’arte *Botsify, Chatfuel, Chatbot.com e Wit.ai, IBM Watson Conversation*, soffrono di comuni limitazioni.

La limitazione più ricorrente, è la mancanza di ramificazioni del dialogo.

L’utente finale ha come unica possibilità la realizzazione di un dialogo lineare senza alcuna variazione in termini di contesto e contenuti.

*Chatfuel*, oltre a limitare il dialogo a una struttura lineare, permette sì di creare bot, ma solo su una piattaforma specifica: *Facebook Messenger*.

*Botsify* è un altro sistema che permette di realizzare chatbot allargando la platea di piattaforme utilizzabili (*Slack* e siti web personali), ma limita la creazione degli stessi a 10 ed è necessario sottoscrivere un abbonamento mensile.

*Chatbot.com* integra le ramificazioni del dialogo, pur limitandolo a un semplice “botta e risposta” fra utente e chatbot stesso. Le conversazioni sono puramente limitate a risposte predefinite, rendendo a tutti gli effetti il bot inabile a recepire frasi più complesse da parte dell’utilizzatore finale.

*Wit.ai* così come *IBM Watson Conversation* superano queste limitazioni, rimanendo tuttavia preclusi agli utenti senza forti competenze informatiche, specialmente legate alla programmazione.

*IBM Watson Conversation* offre la possibilità di effettuare query ad API esterne: possono essere impiegate dal bot per offrire informazioni aggiuntive all’interno del dialogo.

Nonostante ciò, anch’esso soffre di una limitazione: è necessario saper programmare in specifici linguaggi di programmazione per poterli usare.

Un’altra limitazione comune, è la mancanza di un sistema di apprendimento automatico per il bot.

In molti casi gli utenti finali commettono errori di sintassi, questo implica che l’agente non sia in grado di carpire ciò che l’utente vuole comunicare: il bot non è quindi in grado di proseguire nell’evoluzione del dialogo.

Capitolo 3

Soluzione Proposta

Per far fronte alle lacune presenti nel mercato attuale, la soluzione proposta in questa tesi si basa su un ambiente grafico tramite il quale l’utente finale è in grado di produrre un bot tramite elementi visivi per poi essere processati automaticamente al fine di realizzare il chatbot a tutti gli effetti.

L’ambiente grafico è basato su Draw.io, una soluzione web open-source che permette di realizzare grafi, diagrammi, mappe e schemi.

Il chatbot, una volta disegnato tramite Draw.io viene costruito tramite DialogFlow.

DialogFlow è una piattaforma di natural language processing realizzata da Google che permette di realizzare chatbot.

Allo stato dell’arte attuale e rispetto alle soluzioni presenti a mercato esposte nel capitolo precedente, DialogFlow include le seguenti peculiarità:

* **Apprendimento del chatbot**: DialogFlow dispone di un sistema di apprendimento automatizzato volto a migliorare le capacità di comprensione del chatbot stesso.   
  Questa necessità subentra a causa di differenti errori che gli utilizzatori finali possono commettere durante una conversazione con il chatbot.   
  Gli errori in questione sono legati per lo più alla battitura (termini scritti in modo errato), alla pronuncia (espressioni gergali, dialettali, accenti tonici, inflessioni della voce), oppure a frasi espresse in modo incompleto.  
  Il chatbot è in grado di apprendere questi errori a seguito di numerose conversazioni sostenute con gli utenti finali, in modo tale da aumentare le sue capacità di comprensione.
* **Svincolo dalle piattaforme**: i chatbot realizzati con DialogFlow permettono di essere usati sia come soluzione standalone, sia come integrazione per servizi di messaggistica (Telegram, Slack, Facebook Messenger, ecc.), sia come servizio aggiuntivo per siti web o applicazioni di terze parti quali Skype, sia come assistente vocale.   
    
  Quest’ultimo punto è di grande rilievo poiché permette di essere utilizzato con la voce. DialogFlow dà la possibilità di essere abbinato ad assistenti vocali quali Google Home, Alexa e non solo: siamo in grado di integrarlo a sistemi “fai da te” quali Raspberry Pi o altre piattaforme embedded.   
    
  *Tutto ciò permette di fornire supporto a persone con problemi visivi.*
* **Interrogazione di API esterne**: DialogFlow, previa attivazione di un abbonamento a consumo, permette di integrare in totale libertà l’uso di API esterne.   
  Questo permette di integrare al nostro agente informazioni legate a servizi di terze parti, quali ad esempio servizi metereologici, di trasporto e dati legati all’ente stesso che intende usarlo.
* **Flusso di dialogo ramificato**: DialogFlow permette di espandere il contesto della conversazione eliminando la limitazione del dialogo lineare.
* **Poliglottismo**: DialogFlow, una volta realizzato il chatbot, dà la possibilità di replicare i dialoghi in differenti lingue, eliminando le barriere linguistiche se necessario.

Questa piattaforma tuttavia, ha due limitazioni.

Pur offrendo un’interfaccia grafica abbastanza lineare, un utente con scarse competenze informatiche può trovare difficoltà nella creazione del chatbot.

L’integrazione e la conseguente esecuzione di chiamate ad API esterne è possibile tramite la scrittura di un file in JavaScript associato direttamente al chatbot.

Questa limitazione è invalicabile per un utente senza solide competenze informatiche.

Le limitazioni appena esposte vengono superate dalla soluzione proposta in questa tesi.

DialogFlow: come funziona

DialogFlow, come anticipato nella sezione precedente, è un framework acquisito da Google (precedentemente nominato come “api.ai”) che permette la creazione di chatbot intelligenti, d’ora in avanti nominati “agenti” con tutte le peculiarità già esposte.

L’agente all’interno di DialogFlow

L’agente in DialogFlow, non è nient’altro che l’entità automatizzata che provvede a sostenere il dialogo con l’utilizzatore umano, carpire il contesto del dialogo, eventuali parole chiave e fornire risposte in funzione delle esigenze espresse dalla sua controparte reale.

Ogni agente, si compone di diversi elementi, il principale è l’intent.

Intent

L’intent è l’unità elementare del dialogo tra agente ed utente.

A fronte degli input forniti dall’utente, l’agente produrrà un output a seconda di quanto specificato dal creatore dell’agente stesso.

Questo processo avviene all’interno di un intent che si compone di tre elementi principali:

* **Training Phrases**: si tratta di frasi esemplificative scritte dal creatore dell’agente che l’utente finale può scrivere o pronunciare.
* **Azioni e parametri**: le training phrases possono contenere al loro interno parole che possono essere associate a specifiche azioni che l’agente deve eseguire o tipologie di dati.  
  In quest’ultimo caso, possono essere sono di rilievo per l’agente al fine di fornire risposte all’utente finale.
* **Risposte**: si tratta di testi, immagini, tracce audio, contenuti multimediali forniti in risposta all’utente che interagisce con l’agente. L’obiettivo comune delle risposte è guidare nei passi successivi del dialogo l’utente, fornirgli i dati o le informazioni di cui necessita o terminare la conversazione stessa.

Facendo un esempio concreto, l’agente può fornire l’orario del primo treno che partirà dalla stazione A alla stazione B.

L’agente dovrà tenere conto di tre informazioni: l’orario di partenza, la stazione in cui l’utente desidera arrivare e quella da cui parte.

È necessario tenere traccia della stazione di partenza espressa dall’utente per mantenere il flusso di dialogo coerente per due motivi: fornire una risposta finale concreta e valida all’utente e rendere quanto più naturale il dialogo.

Questo esempio esprime ciò che è il contesto. Gli intent infatti, possono avere dei contesti di input e dei contesti di output per far fronte a ciò.

Questo meccanismo, permette di mantenere in memoria i parametri di rilievo carpiti dalle frasi espresse da parte dell’utente finale per tutta la durata del dialogo.

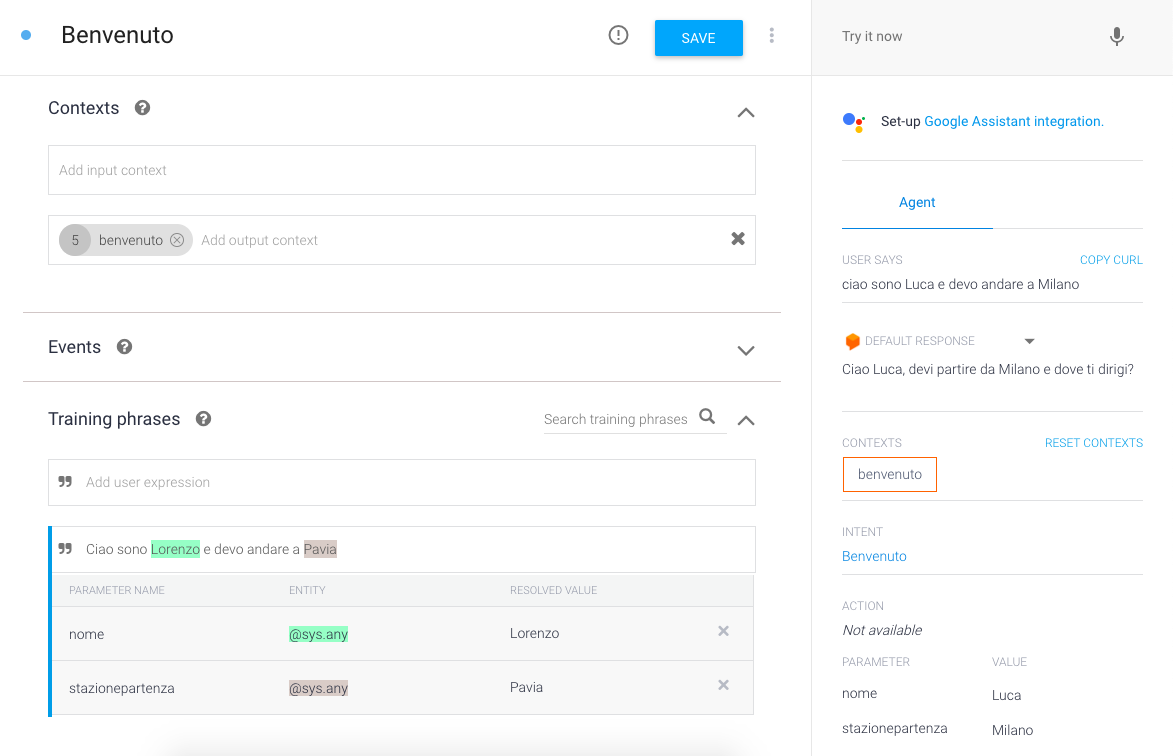


Figura 1: un esempio di intent

Come si evince dall’immagine, ci troviamo di fronte a un Intent il cui nome è “Benvenuto”.

L’intent, ha come contesto di output se stesso: questo perché ci sono due parametri che serviranno per il punto del dialogo successivo, il nome dell’utente e la stazione da cui desidera partire.

Abbiamo infatti la sezione “Training Phrases”: queste, sono le frasi potenziali che l’utente può pronunciare e ad esse, corrisponderà la risposta associata a questo intent.

Nelle training phrases abbiamo modo di specificare quali sono le informazioni di rilievo che l’utente sta offrendo all’agente: nell’esempio le parole “Lorenzo” e “Pavia” sono evidenziate con un rispettivo colore e riportate nella tabella sottostante.

Questa tabella rappresenta ciò che sono i parametri. Essi possono essere nominati con una parola chiave (in questo caso rispettivamente “nome” e “stazionepartenza”) per poi essere impiegati per tre operazioni differenti:

1. Nella risposta diretta all’utente
2. Come dati per chiamate ad API esterne
3. In altri intent purché abbiano nei propri contesti di input quello dell’intent in questione

Di fatto, l’intent ha un’ultima parte:

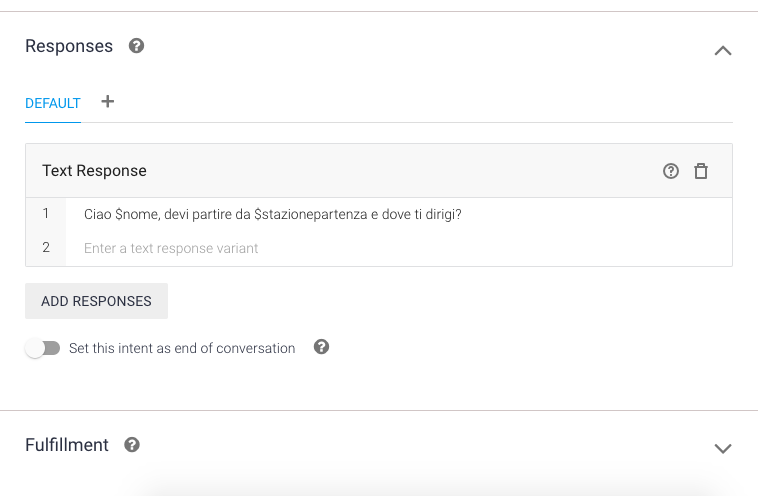


Figura 2: le risposte dell'intent

Abbiamo modo di impostare le risposte dell’agente: esse possono essere testuali, multimediali oppure risultato di una chiamata ad API esterne. Nel caso in esame, abbiamo impostato una semplice frase di testo dove sono indicati i due parametri esplicati in precedenza.

Come infatti si può notare dalla Figura 1, alla frase dell’utente “Ciao sono Luca e devo andare a Milano”, l’agente carpisce le due parole chiave e risponde “Ciao Luca, devi partire da Milano e dove ti dirigi?”.

­­Per comprendere a pieno i contesti, guardiamo la successiva immagine:

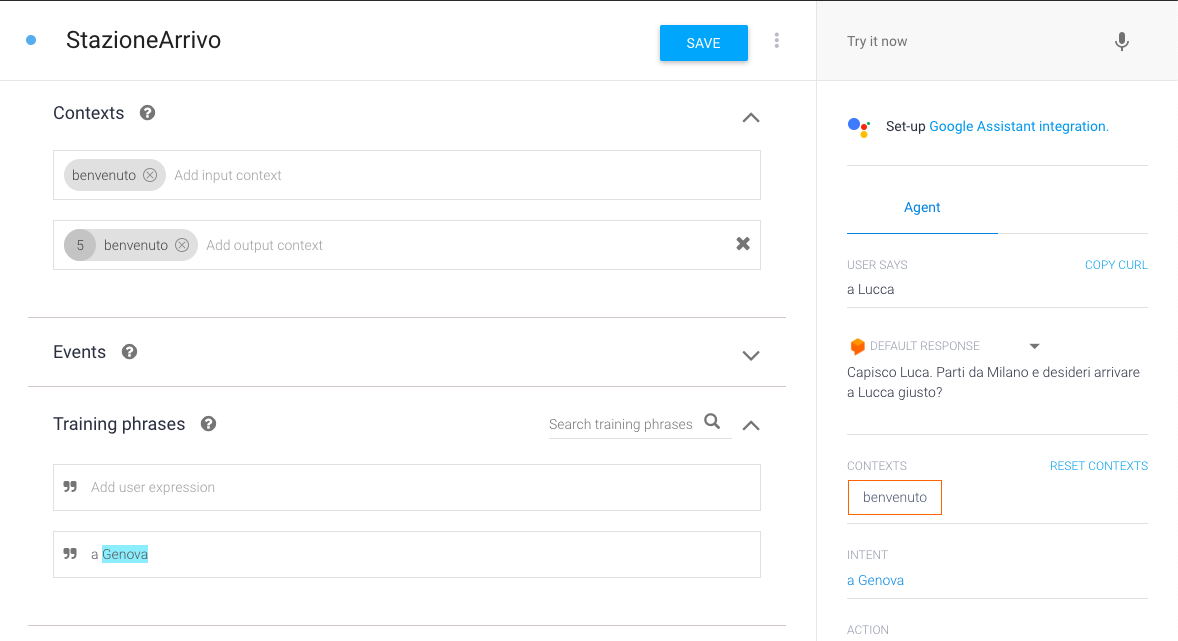


Figura 3: l'intent successivo dell'agente

Questo intent rappresenta il secondo punto del dialogo: ha infatti come contesto di input quello dell’intent esposto in precedenza.

Possiamo sempre notare dall’immagine, che oltre a carpire un altro parametro (ovvero la stazione di arrivo), l’agente risponde all’utente tenendo traccia del suo nome e della sua stazione di partenza: “Capisco Luca. Parti da Milano e desideri arrivare a Lucca giusto?”.

In precedenza, abbiamo detto che, oltre ai contenuti multimediali, l’agente può fornire risposte a seguito dei risultati ottenuti da chiamate ad API esterne: ciò è possibile grazie ai “Fulfillment”

Fulfillment

Un agente creato su DialogFlow permette di invocare API esterne. Queste, tornano utili in diversi scenari, in particolar modo per ottenere informazioni utili all’utente finale.

Ciò che avviene, è la generazione di un file JavaScript in cui si specifica la mappatura degli intent e le chiamate alle API: quest’ultimo punto, avviene tramite protocollo HTTP e l’uso di una callback.

Poiché in diversi casi le chiamate vanno parametrizzate in funzione di ciò che ha espresso l’utente, è possibile ottenere i parametri degli intent o dei contesti coinvolti per le chiamate stesse.

Riprendendo l’esempio espresso in precedenza, una concreta applicazione di questa utilità è invocare un’API di Trenitalia. Una volta ottenute le stazioni del tragitto che intende effettuare l’utente e l’orario in cui desidera partire, possiamo prendere questi tre parametri e invocare il servizio.

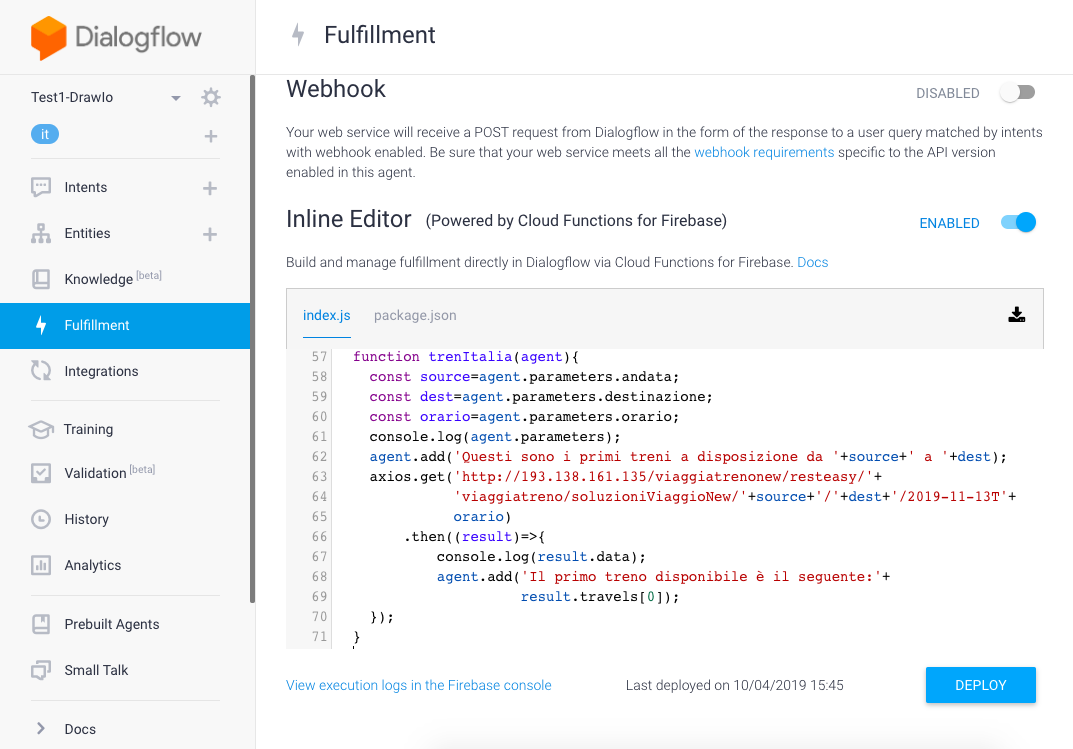


Figura 4: richiesta ad API Trenitalia

Come possiamo notare, nella sezione “Fulfillment” di DialogFlow e dalla funzione “trenItalia”, otteniamo i tre parametri dal contesto conversazionale. Tramite la libreria “axios”, che permette di eseguire richieste HTTP, inseriamo l’URL necessario passando in GET i tre parametri.

Nella callback della chiamata stessa, tramite la funzione “agent.add”, forniamo una risposta all’utente finale tramite l’intent su cui abbiamo effettuato la mappatura:



Figura 5: la mappatura della chiamata API sull'intent

La funzione “intentMap.set”, dato il nome dell’intent e la funzione, permette di mappare la funzione contenente la chiamata HTTP e l’intent stesso: in questo modo possiamo generare risposte provenienti da API esterne all’utente.

Naturalmente questo processo richiede un’esperienza consistente sull’uso di JavaScript e della programmazione, nonché dei protocolli di comunicazione, motivo per cui sarà totalmente trasparente al creatore finale tramite il sistema creato.

Capitolo 4

Architettura della Soluzione Proposta

Stando a quanto esposto nel capitolo precedente, dobbiamo rendere quanto più trasparente possibile il processo di generazione dell’agente ed eliminare la necessità di saper programmare all’utente finale.

Da un punto vista macroscopico, l’architettura della soluzione proposta si compone come segue:



Il secondo, terzo e quarto passo sono trasparenti all’utente finale: una volta “disegnato” l’agente con Draw.io sarà direttamente in grado di utilizzare l’agente nella sua interezza.

Per rendere quanto più minima l’esperienza di programmazione necessaria all’uso di questa architettura, è stato implementato un linguaggio visuale da impiegare per realizzare il grafo rappresentante l’agente.

Affinché l’architettura funzioni, è stato necessario realizzare un interprete in grado di comprendere il contenuto del grafo, estrapolarne le informazioni necessarie per la creazione dell’agente e infine, inoltrare i dati a DialogFlow attraverso le API di Google.

L’interprete, che solo interprete non è, è stato realizzato in Javascript/Typescript con l’ausilio di NodeJS e una libreria sviluppata direttamente da Google.

Infatti non si occupa solo di interpretare il grafo, ma una volta estrapolati i dati utili per costituire l’agente, si interfaccia con le API di Google affinché si possa effettivamente rendere vivo l’agente stesso.

La libreria in questione, è “DialogFlow: Node.js Client”. Essa, permette di interfacciarsi ai servizi di DialogFlow, includendo quindi il nostro agente, attraverso differenti metodi messi a disposizione direttamente mappati alle chiamate HTTP da effettuare a Google.

Di tale libreria, sono stati usati i metodi che permettono la creazione e l’aggiornamento degli intent, cuore pulsante di ogni dialogo.

Limitazioni

Le librerie impiegate non concedono totale margine di libertà e, poiché sono ancora in fase di sviluppo, hanno presentato alcuni bug che, pur non limitando il raggiungimento del nostro obiettivo, ci hanno costretto ad aggirare i problemi.   
In aggiunta a ciò, in questi mesi Google sta effettuando l’upgrade delle API di DialogFlow e migrando tutti i servizi nonché endpoint messi a disposizione.

Il primo limite, è stato quello della creazione dell’agente: esso, non può essere attualmente creato “ex novo”. È necessario crearne uno vuoto accedendo al proprio account sul sito ufficiale di DialogFlow.

Solo a questo punto, è possibile utilizzare le API di Google e quindi il sistema realizzato.

Un'altra limitazione, è stata il riconoscimento dei parametri. Pur forzando tramite le chiamate API gli attributi dei parametri, cosa necessaria dal momento che vanno creati, ce n’è uno in particolare che specifica se il nostro parametro è una lista oppure no.

Nonostante in fase di creazione e aggiornamento si specifica che tutti i parametri dell’agente non sono liste, le API li vedono come tali.

Google permette sì di ricorrere all’uso di API esterne e il sistema realizzato altrettanto, permettendo agli utenti di includere chiamate ad API esterne senza saper programmare in JavaScript.

Google tuttavia, pur permettendo di ricorrere a questa funzionalità, richiede un account a cui sia associato un metodo di pagamento. Il motivo è molto semplice: trattandosi di chiamate HTTP a servizi esterni, si va a consumare traffico dati, di conseguenza è necessario attivare un piano dati a consumo direttamente tramite Google.

Affinchè l’utente finale possa fare un uso completo del sistema, è necessario integrare le sue credenziali Google tramite un file JSON e un ID di sessione ottenibile direttamente dalla console di DialogFlow. Entrambe le procedure vanno eseguite manualmente poiché senza questi dati sarebbe impossibile interfacciarsi con le API di Google e quindi creare l’agente.

Come già esposto in precedenza, per includere API di terze parti nell’agente, è necessario attivare un sistema di pagamenti associato al proprio account Google. Questo servizio, poiché implica consumo dati, si basa su una politca “pay per consume”: in base al numero di richieste effettuate alle API esterne, Google ci addebiterà il costo.

Le risposte di eventuali API esterne, devono essere fornite in JSON ben formato: DialogFlow non accetta altri formati per ottenere risposte da esse.

Linguaggio Visuale - Introduzione

Il sistema realizzato ha come obiettivo principale di permettere la realizzazione di agenti a coloro che non sanno sviluppare codice.

Per tanto, ho realizzato un linguaggio visuale attraverso l’uso di Draw.io: un software gratuito online che permette la realizzazione di grafi.

Questo, permetterà poi la generazione di un file XML: verrà impiegato dall’interprete per estrapolare tutti i componenti che costituiranno l’agente finale, per poi crearlo effettivamente.

Il linguaggio si basa su quattro elementi principali che costituiscono qualsiasi agente: gli intent, i contesti, le credenziali di accesso e i fulfillment.

Linguaggio Visuale - Intent

L’intent viene costruito con il seguente elemento:

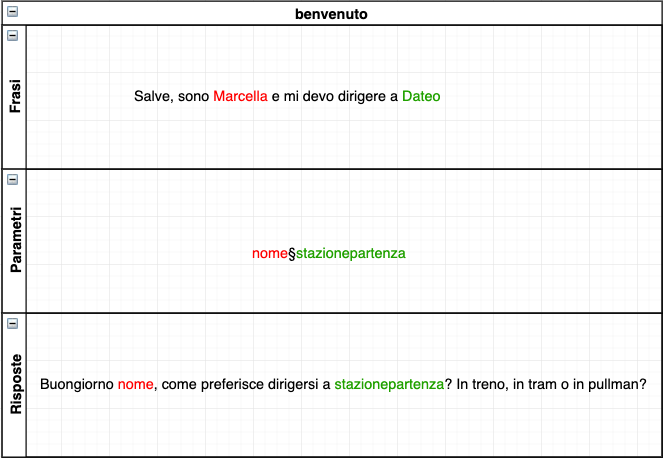


Figura 6: un intent realizzato con il linguaggio visuale

Esso segue la logica di DialogFlow, ha di fatto tre sezioni e un nome (in questo caso “benvenuto”). Le sezioni rappresentano rispettivamente:

1. Frasi: sono le frasi che l’utente può pronunciare durante il dialogo. Nel caso in esempio, troviamo “Salve, sono Marcella e mi devo dirigere a Dateo”.
2. Parametri: sono i parametri a cui associare le parole chiave da carpire all’utente.
3. Risposte: contiene le risposte testuali impiegate dall’agente per rispondere all’utente.
4. API: utilizzata in sostituzione alla sezione “Risposte”. Essa conterrà il nome dell’API che si intende invocare

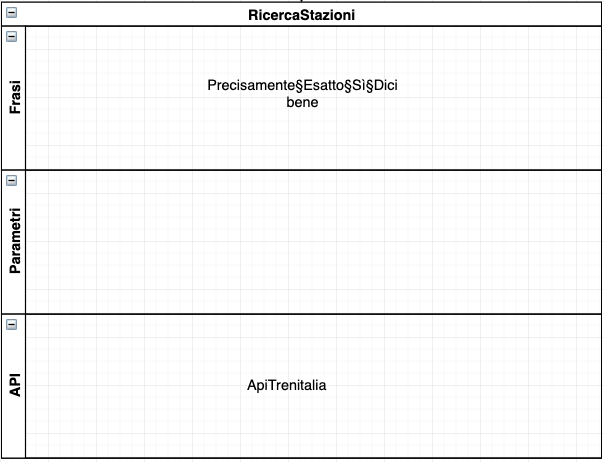


Figura 7: un intent in cui avviene una chiamata ad un API

Linguaggio Visuale – I parametri

Al fine di rendere completa la possibilità di creare gli intent, quindi associare i parametri alle parole chiave delle frasi espresse dall’utente e usarli poi in seguito nelle risposte o nelle API dell’agente, viene fatto l’uso dei colori.

Riprendendo l’esempio presente in figura 6, il nome dell’utente (ovvero “Marcella”), lo si vuole associare al parametro “nome”: quest’ultimo dev’essere colorato con un colore scelto liberamente dall’utilizzatore del sistema.

Affinché l’associazione venga poi rispettata in fase di creazione dell’intent, la parola chiave presente nella trainingphrase (quindi “Marcella”) deve essere colorata con lo stesso colore usato in precedenza.

In questo modo, permettiamo di integrare nell’agente i parametri carpiti dalle frasi pronunciate dall’utente: dobbiamo permettere di utilizzarli nelle risposte. Per far fronte a ciò, è sufficiente impiegare lo stesso colore usato per il parametro in quesitone e utilizzare il nome impostato per il parametro stesso all’interno della risposta.

Nel caso presente in figura 6 infatti, noteremo che la risposta dell’intent “benvenuto” sarà “Buongiorno nome…”.

In diversi casi, i parametri di un intent possono essere utilizzati per fornire risposte in altri intent e, naturalmente, come parametri per invocare API esterne.



Figura 8: uso di parametri provenienti da intent differenti

Per far fronte a ciò, è sufficiente procedere come esposto in figura 8, rispettando la seguente sintassi:

**nomeIntent.nomeParametro**

Nell’intent “viaggiotreno” vogliamo impiegare all’interno della risposta i due parametri “nome” e “stazionepartenza” impostati nell’intent “benvenuto”. Tenendo presenti le colorazioni in figura 6 e impiegandole nell’intent “viaggiotreno” dovremo appunto scrivere “benvenuto.stazionepartenza” e “benvenuto.nome”.

Questo però non basta: sono parametri provenienti da un altro intent, di conseguenza dobbiamo renderli accessibili tramite il contesto di dialogo.

Linguaggio Visuale – Contesti di dialogo

I contesti di dialogo, come già visto in precedenza, permettono di creare non solo un flusso, ma di tener traccia di tutto quello che l’utente ha espresso all’agente, incluse le parole chiave (i parametri).

Poiché si tratta di un flusso, per connettere due intent è sufficiente usare una freccia.



Figura 9 creazione di un contesto

In questo modo, l’intent “viaggiotreno” avrà come contesto di input “benvenuto”, a loro volta, gli intent connessi con la freccia a “viaggiotreno”, avranno come contesto di input sia “benvenuto”, sia “viaggiotreno”.

Il meccanismo appena descritto, permette di utilizzare i parametri creati in altri intent e la realizzazione di un flusso di dialogo.

In DialogFlow abbiamo visto che in contesti di input e output possono essere creati a discrezione dell’utente. Per semplicità d’uso questo passaggio sarà trasparente all’utente finale.

Di fatto, qualsiasi intent che ha almeno una freccia uscente, avrà automaticamente impostato un contesto di output con il suo stesso nome (quindi l’intent “benvenuto” avrà come contesto di output “benvenuto”).

Allo stesso modo, un intent che ha una freccia entrante, avrà come contesti di input tutti i contesti di output dell’intent a cui è connesso (“viaggiotreno” ha come contesto di input “benvenuto”).

Seguendo questa logica e l’esempio guida mostrato fino ad ora, se si collegasse un terzo intent a “viaggiotreno”, accadrebbe ciò:

* “viaggiotreno” imposterebbe come contesti di output lui stesso più quelli provenienti dalla freccia entrante (ovvero “benvenuto”).
* Il terzo intent avrebbe come contesti di input sia “viaggiatreno”, sia “benvenuto”.

Questo meccanismo, è necessario per creare un automatismo totalmente trasparente nella creazione dei contest idi dialogo.

Linguaggio Visuale – Gestione parametri e frasi multiple

L’utente finale naturalmente, per completezza d’uso del sistema, può inserire quanti parametri, frasi e risposte desidera all’interno di un singolo intent.

Frasi e risposte tuttavia, contengono elementi di punteggiatura, di conseguenza è stato necessario individuare un simbolo di scarso utilizzo che potesse permettere all’interprete di capire come separarle.

In figura 6, noteremo che nelle frasi dell’intent è riportata la seguente stringa:

**“Precisamente§Sì§Esatto§Dici bene"**

Per inserire più di una trainingphrase, parametro o frase di risposta, è sufficiente separarle con il carattere “§”. In questo modo sarà possibile gestire più di un singolo elemento all’interno dell’intent.

Linguaggio Visuale – Gestione credenziali

Come specificato nelle limitazioni delle API di DialogFlow, è necessario creare un file JSON e ottenere dei parametri di sessione affinché il sistema funzioni.

I due parametri sono rispettivamente il nome del progetto per far capire a DialogFlow su quale agente vogliamo intervenire e un sessionID che permette di invocare i servizi.



Figura 10 ottenimento dati sessione

Nella sezione a destra della console di DialogFlow, possiamo testare il nostro agente. Scrivendo una frase qualsiasi, a seguito della risposta dell’agente, apparirà la voce “COPY CURL”.

Cliccando, il contenuto è impostato come segue:

**curl -H "Content-Type: application/json; charset=utf-8" -H "Authorization: Bearer ya29.c.Kl6xBw8cTQHKE1igj99wI92\_rR3grgDgsOVi-7S47RNQM7VjJ-056U4-d5dmkVf1ilL2XHiPtdt2A3IOAQon7OnceJ9HyZf9GpHGzlr7GUsE7cEVbgGtaeOdclkULeNg" -d "{\"queryInput\":{\"text\":{\"text\":\"Ciao\",\"languageCode\":\"it\"}},\"queryParams\":{\"timeZone\":\"Europe/Rome\"}}" "https://dialogflow.googleapis.com/v2beta1/projects/test1-drawio-fsilir/agent/sessions/6bbd5860-a8d6-0e3d-675a-7b232b06d613:detectIntent"**

L’ultima sezione è un URL: da questo dobiamo estrapolare le due informazioni di rilievo.

Nella fattispecie: [https://dialogflow.googleapis.com/v2beta1/projects/**test1-drawio-fsilir**/](https://dialogflow.googleapis.com/v2beta1/projects/test1-drawio-fsilir/)

La parte in grassetto, è il nome del progetto.

Proseguendo: /agent/sessions/**6bbd5860-a8d6-0e3d-675a-7b232b06d613**:

È il sessionID.

Questi due parametri, verranno inseriti nel grafo dell’agente come segue:



Figura 11 le credenziali del chatbot

Tramite la dicitura “ID: idAgente Chiave: chiaveDiSessione”, evitiamo all’utente finale di inserirli manualmente all’interno dell’interprete ogni qual volta è intenzionato a farne uso.

Linguaggio Visuale – API

Per poter invocare un’API, è necessario realizzare il seguente elemento grafico:

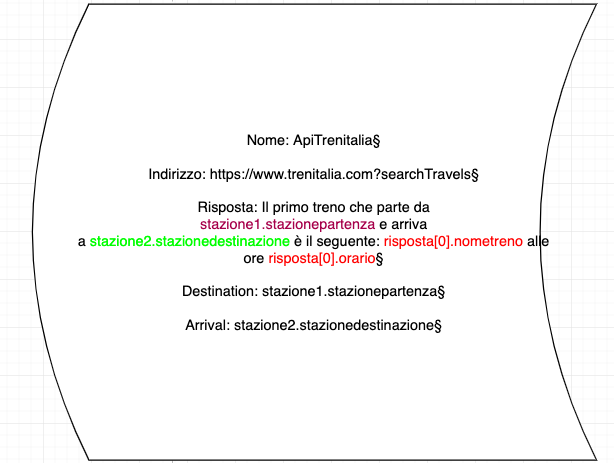


Figura 12: realizzazione di una chiamata ad API esterna

Esso si compone come segue:

1. Nome: è il nome di riferimento per riconoscerla
2. Indirizzo: è l’URL che si intende invocare
3. Risposta: è la risposta che l’agente deve fornire all’utente finale a seguito della chiamata effettuata all’API esterna.
4. Parametri: si tratta dei parametri da impiegare per effettuare la chiamata stessa.

Ognuna delle quattro sezioni termina con il carattere speciale “§” per far capire all’interprete dove terminano.

Linguaggio Visuale – I parametri delle API

Per invocare un’API esterna, è necessario spesso e volentieri impostare dei parametri: in molte situazioni questi derivano direttamente dal contesto di dialogo dell’agente.

Seguendo le direttive d’uso dell’API in questione, è sufficiente impostarle come segue:

**NomeParametroAPI: valore§ | parametroAgente§**

Stando alla sintassi e all’esempio in figura 12, per invocare l’API di Trenitalia, è necessario passare “Destination” come parametro GET ed è nostra intenzione valorizzarlo con il contenuto del parametro “stazionepartenza” presente nell’intent “stazione1”.

Naturalmente, per offrire una libertà d’uso completa all’utente finale, si possono inserire quanti parametri si desidera o quanti sono necessari per invocare l’API desiderata.

Linguaggio Visuale – Le risposte delle API

Un’API viene associata a un intent, di conseguenza sarà necessario offrire una risposta all’utente finale a seguito della chiamata della stessa.

L’utente ha libera discrezione: può usare testo libero, può usare parametri derivanti dal contesto di dialogo (purché colorati, come si vede in figura 12 con l’uso del parametro “stazione1.stazionepartenza”), ma soprattutto può usare i contenuti della risposta data dall’API.

Per quest’ultimo punto, è sufficiente usare la parola chiave “risposta” e colorarla: questa, non è nient’altro che la variabile ottenuta dalla callback a seguito della chiamata HTTP effettuata nel fulfillment dell’agente.

In funzione del contenuto (nel caso di figura 12, è un array e a noi interessa la sua prima cella) e della struttura di risposta, lasciamo libero arbitrio all’utilizzatore finale.

Il tutto è predisposto, come da limitazioni esposte, per oggetti JSON ben formati. Nell’esempio di figura 12 infatti, usiamo la zeresima cella di un array i cui attributi sono rispettivamente “nometreno” e “orario”: segue infatti la stessa identica sintassi impiegata per leggere gli oggetti JSON.

Interprete – Introduzione

Draw.io come esposto in precedenza, permette di esportare i grafi creati in formato XML. Questi, affinché possano essere letti, estrapolate le informazioni di rilievo e inviati a Google, necessitano di essere interpretati.

Attraverso due librerie per NodeJS, è stato possibile realizzare un sistema di lettura del grafo: “libxmljs” e “xml2js”.

Seguendo la logica di costruzione di un agente, è necessario estrapolare tutte le informazioni legate agli intent, quindi le trainingphrases, i parametri e le risposte, i contesti di dialogo, le chiamate ad API esterne, nonché le credenziali dell’agente.

Ogni elemento che costituisce il grafo e quindi l’XML generato da Draw.io, ha un identificativo univoco e questo ha permesso di eseguire un’interpretazione corretta e completa del file.

Interpretando ogni elemento XML, siamo in grado di costituire quelli che saranno gli oggetti che formano un intent nella sua interezza, i contesti e il fulfillment qual ora l’agente si costituisse anche di chiamate API

Interprete – Interpretazione degli Intent

Un intent, come abbiamo visto nella sezione relativa al linguaggio visuale, si compone di un unico oggetto, costituito a sua volta da quattro entità XML, tutte di tipo “mxCell”:

<mxCell id="m0-oef0f0qb-4m2KXaeG-1" value="Benvenuto" style="swimlane;html=1;childLayout=stackLayout;resizeParent=1;resizeParentMax=0;horizontal=1;startSize=20;horizontalStack=0;" parent="1" vertex="1">

<mxGeometry x="70" y="70" width="520" height="380" as="geometry"/>

</mxCell>

La prima, è ovviamente quella che identifica l’intent stesso. È identificabile tramite l’attributo “style”: al suo interno troveremo come parole chiave “swimlane” e “stackLayout”. Come si evince dall’esempio, il contenuto dell’attributo “value” è nient’altro che il nome dell’intent.

Le tre sezioni dell’intent, ovvero le frasi, i parametri e le risposte, sono organizzate come segue:

<mxCell id="m0-oef0f0qb-4m2KXaeG-2" value="Frasi" style="swimlane;html=1;startSize=20;horizontal=0;" parent="m0-oef0f0qb-4m2KXaeG-1" vertex="1">

<mxGeometry y="20" width="520" height="120" as="geometry"/>

</mxCell>

<mxCell id="hRAsFazEqZHqzmIs6ujT-2" value="Ciao sono &lt;font color=&quot;#ff0000&quot;&gt;Lorenzo&lt;/font&gt;§Buongiorno mi chiamo &lt;font color=&quot;#ff0000&quot;&gt;Andrea&lt;/font&gt;" style="text;html=1;strokeColor=none;fillColor=none;align=center;verticalAlign=middle;whiteSpace=wrap;rounded=0;" parent="m0-oef0f0qb-4m2KXaeG-2" vertex="1">

<mxGeometry x="110" y="50" width="320" height="20" as="geometry"/>

</mxCell>

Nell’esempio riportato, noteremo il primo oggetto “mxCell” il cui valore è “Frasi”, il secondo invece, contiene due frasi con due parole chiave colorate (ovvero “Lorenzo” e “Andrea”), identificabili tramite l’attributo “font”.

Entrambi gli oggetti hanno un attributo nominato “parent”: è l’identificativo univoco che Draw.io genera per ogni entità XML del grafo.

Come possiamo notare dall’esempio, il terzo oggetto XML ha come “parent” l’id del secondo, a sua volta il secondo ha come “parent” l’id del primo.

In questo modo, l’interprete sa che il “value” contenuto nel terzo oggetto, sono le training phrases (ovvero le frasi inserite nella sezione “Frasi” dell’intent) dell’intent “benvenuto”. Naturalmente, il secondo oggetto è necessario all’interprete per capire che si tratta esattamente delle training phrases da inserire nell’intent.

Questo meccanismo di riconoscimento, avviene anche per le altre due sezioni: i parametri e le risposte, oppure l’associazione a un’API di terze parti.

else if (graphXMLCells[c].ATTR.style.indexOf("childLayout")>=0){

//Intent/Fulfillment

var valueXML=graphXMLCells[c].ATTR.value;

valueXML.replace("&lt;","<");

valueXML.replace("&gt;",">");

valueXML.replace("&quot;","\"");

valueXML.replace("&amp;","\"");

//Nome

var name=valueXML;

//Sezione 1: frasi

var phraseString="";

graphXMLCells.forEach((cell)=>{

if (cell.ATTR.parent==graphXMLCells[c].ATTR.id){

if (cell.ATTR.value=="Frasi"){

var finalFrasi="";

graphXMLCells.forEach((cellFrasi)=>{

if (cellFrasi.ATTR.parent==cell.ATTR.id){

finalFrasi=cellFrasi.ATTR.value;

}

});

phraseString=finalFrasi;

//Rimuoviiamo entità HTML in eccesso

phraseString=phraseString.replace(/<\/?span[^>]\*>/g,"");

phraseString=phraseString.replace(/<\/?p[^>]\*>/g,""); phraseString=phraseString.replace(/<\?br[^>]\*>/g,"");

phraseString=phraseString.replace(/&nbsp;/g,"");

}

}

});

In questa sezione di codice, notiamo come avviene l’interpretazione della sezione “frasi” una volta riconosciuto un intent all’interno della lettura del grafo. Ottenuta la cella relativa alle frasi vere e proprie, queste vengono analizzate e pulite da entità HTML in eccesso che non sono rilevanti ai fini di realizzazione dell’intent nonché della sintassi e della grammatica delle frasi stesse.

Le frasi, una volta ottenute, vanno associate correttamente ai parametri dell’intent nonché alle risposte, motivo per cui esse vengono mappate con i parametri:

stringParameters.split("§").forEach(element => {

if (element.length>0)

parameters.push(getParameterByXML(element));

});

var phrases=[];

phrases=phraseString.split("§");

phrases=phrases.map((phrase)=>{

return splitTrainingPhrase(phrase,parameters);

});

Il processo di mappatura tra parametri e frasi, avviene anche per le risposte dell’intent stesso.

Interprete – Interpretazione delle API

A differenza di quanto avviene per le risposte testuali e quindi, la mappatura dei parametri sulle stesse, l’interprete cerca di capire se l’intent ha una chiamata ad un’API anziché una risposta tradizionale:

var hasAPI=0;

graphXMLCells.forEach((cell)=>{

if (cell.ATTR.parent==graphXMLCells[c].ATTR.id){

if (cell.ATTR.value=="API"){

hasAPI=1;

//Prendiamo il nome dell'api richiesta

graphXMLCells.forEach((cellAPI)=>{

if (cellAPI.ATTR.parent==cell.ATTR.id){

intentAPIRequired.push({"intentName":name.toLowerCase(),"APIRequired":cellAPI.ATTR.value});

}

});

}

}

});

In questo caso, anziché provvedere alla mappatura dei parametri, si provvede ad associare all’intent in questione l’API desiderata.

Quest’ultima, viene identificata tramite l’attributo “dataStorage” presente nel relativo oggetto XML:

else if (graphXMLCells[c].ATTR.style.indexOf('dataStorage')>=0){

var stringAPI=graphXMLCells[c].ATTR.value;

stringAPI=stringAPI.replace(/<\/?div[^>]\*>/g,"");

stringAPI=stringAPI.replace(/<\/?br[^>]\*>/g,"");

stringAPI=stringAPI.replace(/<\/?img[^>]\*>/g,"");

stringAPI=stringAPI.replace(/<\/?span[^>]\*>/g,"");

stringAPI=stringAPI.replace(/<\/?p[^>]\*>/g,"");

stringAPI=stringAPI.replace(/&nbsp;/g," ");

var nameAPI=stringAPI.substr(0,stringAPI.indexOf("§"));

nameAPI=nameAPI.replace(/<\/?font[^>]\*>/g,"");

nameAPI=nameAPI.substr(5,nameAPI.length-5).trim();

stringAPI=stringAPI.substr(stringAPI.indexOf("§")+1,stringAPI.length-(stringAPI.indexOf("§")+1));

var indirizzo=stringAPI.substr("Indirizzo: ".length,stringAPI.indexOf("§")-"Indirizzo: ".length);

indirizzo=indirizzo.replace(/<\/?font[^>]\*>/g,"");

stringAPI=stringAPI.substr(stringAPI.indexOf("§")+1,stringAPI.length-(stringAPI.indexOf("§")+1));

if (stringAPI.indexOf("</font>")>=0){

stringAPI=stringAPI.substr("</font>".length,stringAPI.length-("</font>".length));

}

Una volta identificato, si provvede a interpretare il contenuto nell’attributo “value”: esso conterrà tutte le informazioni esposte nella sezione relativa al linguaggio visuale.

Affinché l’utente non abbia la minima necessità di agire sul file fulfillment dell’agente e quindi scrivere codice Javascript, l’interprete si occupa di creare da zero il file stesso a seconda di quanto estratto dal grafo:

var fs=require('fs');

fs.writeFile('./fulfillmentBOT.js', stringFile, function(err) {

if(err) {

return console.log("Errore in fase di creazione file fulfillment: "+err);

}

console.log("File fulfillment creato. Copiarne il contenuto in DialogFlow dal file presente in questa cartella: fulfillmentBOT.js.");

});

Infatti, come suggerisce la linea scritta a console, il suo unico compito è copiaincollare il contenuto del file indicato all’interno della sezione “Fulfillment” presente sulla console di DialogFlow.

Interprete – Interpretazione dei contesti

L’interpretazione degli intent e le sue parti, come visto nella sezione precedente, non è sufficiente: bisogna interpretare i contesti e integrarli negli intent stessi.

Da un punto di vista visuale, i contesti sono nient’altro che le frecce connettive tra gli elementi che rappresentano gli intent, all’interno del file XML, sono anche esse oggetti:

<mxCell id="0WU0IiVzRddJUgDPFvJG-23" style="edgeStyle=orthogonalEdgeStyle;rounded=0;orthogonalLoop=1;jettySize=auto;html=1;exitX=0.5;exitY=1;exitDx=0;exitDy=0;entryX=0.5;entryY=0;entryDx=0;entryDy=0;" edge="1" parent="1" source="0WU0IiVzRddJUgDPFvJG-6" target="0WU0IiVzRddJUgDPFvJG-16">

<mxGeometry relative="1" as="geometry"/>

</mxCell>

L’interprete le identifica tramite la stringa “edgeStyle” contenuta all’interno dell’attributo “style”, oltre ad esso ci sono altri due attributi che sono essenziali per capire quali intent connette: “source” e “target”.

Questi ultimi due, sono autoesplicativi: “source” è l’id dell’oggetto XML che identifica l’intent da cui parte la freccia, mentre “target” l’id dell’intent a cui la freccia arriva.

if (graphXMLCells[c].ATTR.style.indexOf('edgeStyle')>=0){

//Freccia

var followup=new ClassfollowupXML();

followup.father=graphXMLCells[c].ATTR.source;

graphXMLCells.forEach((cell)=>{

if (cell.ATTR.id==followup.father){

followup.father=cell.ATTR.parent;

return;

}

});

followup.son=graphXMLCells[c].ATTR.target;

//Aggiungiamo

followups.push(followup);

I contesti, siano essi di input o di ouput, vanno nominati nello stesso modo in cui lo sono gli intent di appartenenza.

if (inoutrow==null){

//Vuol dire che me la devo creare: è un intent senza output

//Perciò, mi prendo l'output del padre

var outPutFather=inouttable.filter((rowFather)=>{return rowFather.id==row.input})[0].output;

//rimuovo me stesso

outPutFather=outPutFather.filter((out)=>{return out!=row.idIntent}); inouttable.push({"id":row.idIntent,"output":[],"input":outPutFather});

}else{

//Rimuovo me stesso dagli input

finalInput=finalInput.filter((final)=>{return final!=inoutrow.id});

inoutrow.input=finalInput;

inoutrow.output=inoutrow.input;

inoutrow.output.push(inoutrow.id);

inoutrow.input=inoutrow.input.filter((inc)=>{return inc!=inoutrow.id});

}

Interprete – Costituzione dell’Intent

Una volta interpretato il grafo e quindi costituiti gli intent nella loro interezza, è necessario creare degli oggetti che rispettino le direttive delle API Google:

export interface Intent {

name?: string;

displayName: string;

webhookState?: string;

priority?: number;

isFallback?: boolean;

mlEnabled?: boolean;

inputContextNames?: string[];

events?: string[];

trainingPhrases?: TrainingPhrase[];

action?: string;

outputContexts?: Context[];

resetContexts?: boolean;

parameters?: Parameter[];

messages?: Message[];

defaultResponsePlatforms?: string[];

rootFollowupIntentName?: string;

parentFollowupIntentName?: string;

followupIntentInfo?: FollowupIntentInfo[];

}

Come notiamo dall’interfaccia in questione, creiamo un oggetto che la rispecchi a seguito dell’interpretazione del grafo:

var intentXML=new ClassintentXML();

intentXML.id=graphXMLCells[c].ATTR.id;

intentXML.name=name.toLowerCase();

intentXML.trainingPhrases=phrases;

intentXML.risposte.push(risposte);

intentXML.parameters=parameters;

intentXML.inputContexts=[];

intentXML.outputContexts=[];

if (hasAPI==1){

intentXML.WebHookState="WEBHOOK\_STATE\_ENABLED\_FOR\_SLOT\_FILLING";

}else{

intentXML.WebHookState="WEBHOOK\_STATE\_UNSPECIFIED";

}

L’if che vediamo e che completa la costituzione dell’intent, serve a specificare se è associata un’API di terze parti oppure no all’intent in questione.

Gli oggetti in questione, una volta creati e, una volta analizzati i contesti, andiamo ad integrarli nel nostro agente finale:

agentXML.setIntents(intentsXML);

agentXML.setFollowups(followups);

Interprete – Creazione dell’agente

Ottenuto l’agente nella sua interezza, dovremo inoltrare delle richieste alle API di Google: come esposto nelle limitazioni, dovremo eseguirle in due fasi.

La prima, consiste nella creazione dell’intent:

var botFile='chatbot.xml';

const agentXMLCreator=require('./parserGraph');

var agentXML;

console.log("Inizio il parsing...");

agentXML=agentXMLCreator.parseGraph(botFile);

const projectId = agentXML.ProjectID;

const agentPath = agentClient.projectAgentPath(projectId);

console.log("Agente creato. Caricamento dati in corso...");

//Per ogni intent parsato dal grafico, lo andiamo ad aggiungere al nostro agente

var intentcreated=[];

agentXML.intents.forEach((intent)=>{

createIntent(projectId,intent.name,intent.id,intent.trainingPhrases,intent.risposte,intent.followUps,intent.parameters, intent.WebHookState)

.then((response)=>{

//Verifichiamo se l'intent è stato creato. In tal caso, dobbiamo integrare i contesti del dialogo in un update

intentcreated.push(response[0]);

if (Array.isArray(response)){

A seguito delle prime righe di codice, necessarie all’interpretazione del grafo vista alla sezione precedente, eseguiamo una funzione di nome “createIntent”: passeremo l’ID dell’agente, la chiave di sessione e tutti gli attributi relativi all’intent. Questa, verrà invocata tante volte quanti sono gli intent realizzati.

const createIntentRequest = {

parent: agentPath,

intent: intent,

};

// Creiamo l'intent

try{

const responses = await intentsClient.createIntent(createIntentRequest);

responses[0].trainingPhrases=intent.trainingPhrases;

responses[0].messages=intent.messages;

return responses;

}

catch (e){

return e;

}

A seguito della risposta ottenuta dalle API di Google, procediamo in due modi differenti:

* Creazione avvenuta con successo: comunichiamo l’esito dell’operazione all’utente e procediamo a seguito con l’integrazione dei contesti.
* Creazione fallita: essa fallisce per due motivazioni principali. Si possono presentare errori di comunicazione (connessione a Internet interrotta ad esempio) oppure l’intent esiste già.

Arrivati a questo punto infatti, se la creazione si è conclusa con successo, integriamo i contesti dell’intent:

if (Array.isArray(response)){

console.log(response[0].displayName+" Creato.");

if ((intent.inputContexts.length>0)||(intent.outputContexts.length>0)){

updateIntent(response[0],intent).then((response)=>{

console.log("Contesti per l'intent "+response[0].displayName+" integrati.");

});

}else{

console.log("L'intent "+response[0].displayName+" non ha contesti di input/ouput da integrare.");

}

Terminando di fatto la creazione dell’agente.

Capitolo 5 – Conclusioni e Analisi

Bibliografia

* D. Fichter e J. Wisniewski, «Chatbots introduce conversational user interfaces,» *Online Searcher,* vol. 41(1), pp. 56-58, 2017.
* Valtolina, Stefano; Barricelli, Barbara Rita; Di Gaetano, Serena (2019): Communicability of traditional interfaces VS chatbots in healthcare and smart home domains. In: *Behaviour & Information Technology,* 195, pp. 1–25. DOI: 10.1080/0144929X.2019.1637025
* Følstad e P. Brandtzæg , «Chatbots and the new world of HCI,» *Interactions,* vol. 24, n. 4, pp. 38-42, 2017.
* Google, «What is conversation design?,» [Online]. Available: https://designguidelines.withgoogle.com/conversation/conversation-design/what-is-conversation-design.html#. [Consultato il giorno 01 Ottobre 2019].
* Manychat, «Flow Builder,» [Online]. Available: http://flowbuilder.manychat.com/. [Consultato il giorno 01 Ottobre 2019].
* DialogFlow: Node.js Client [Online]. Available: <https://github.com/googleapis/nodejs-dialogflow/>